

UNIVERSITY OF SOUTH ALABAMA  
SCHOOL OF COMPUTER & INFORMATION SCIENCES

APPROXIMATION OF A BÉZIER CURVE WITH A MINIMAL  
NUMBER OF LINE SEGMENTS

BY

Athar Luqman Ahmad

A Thesis

Submitted to the Graduate Faculty of the  
University of South Alabama  
in partial fulfillment of the  
requirement for the

Master of Science  
in  
Computer Science

May, 2001

Approved:

---

Chair of Thesis Committee: Dr. Thomas F. Hain Date

---

Member of Committee: Dr. Michael Doran Date

---

Member of Committee: Dr. Fwu-Shan Shieh Date

---

Dean of School of Computer & Information Sciences: Dr. David L. Feinstein Date

---

Director of Graduate Studies: Dr. Roy Daigle Date

---

Dean of Graduate School: Dr. J. L. Wolfe Date

Approximation of a Bézier Curve with a Minimal Number of Line Segments

A Thesis

Submitted to the Graduate Faculty of the  
University of South Alabama  
in partial fulfillment of the  
requirement for the  
Master of Science  
in  
Computer Science

by

Athar Luqman Ahmad  
B. S., University of South Alabama  
Mobile, AL, December, 1997

School of Computer & Information Sciences  
University of South Alabama

May, 2001

To my Mom and Dad whose prayers have been answered by this accomplishment.

## ACKNOWLEDGEMENTS

First of all I am heartily thankful to Dr. Thomas F. Hain for his proposal and introduction of this research topic to me. His motivation and guidance kept me steadfast through every step of the research process from the introduction of the topic to the final defense and compilation of the thesis paper. He is a true master of the field of computer graphics and was an excellent teacher in explaining and making me understand the problem. I will always pay tribute his unconditional help in every part of the research i.e., solving the main problem, literature search, creating of the presentation slides and writing of the thesis paper. His paper and presentation reviews were very extensive and it always seemed like that he has read and corrected each and every single sentence.

Dr. Fwu-Shan Shieh, who is the member of the committee, is one of the most considerate and understanding person I have ever met. Despite his busy schedule he made time to attend the committee meetings and gave very constructive criticism on the presentation slides and as well as the thesis paper. His work experience is directly related to the field of computer graphics and especially printing graphics. He raised good questions that helped me relate the problem to the industry and help me prepare for the thesis defense.

I am also thankful to Dr. Michael Doran, the committee member, who played an important role in the success of this paper. He was a great support and gave constructive feedback during the research.

I would also like to say that without the constructive criticism of Dr. Roy Daigle and his guidelines through the research methodology class this paper would not have been a success. He introduced and taught me to the researching techniques and the process of

research and thesis writing. He is also a great motivator who helped me in making the decision of doing the thesis as appose to comprehensive examination.

Lastly, I cannot ignore my parents, fiancée, family members, close friends, and work colleges who prayed and motivated me all through this process, and I am greatly thankful to each one of them.

## PREFACE

This paper is written of the wide range of readers. The discussed problem is directly related to Computer Graphics but mathematicians who do not have a background in this field can still understand the relationship of this problem with computer graphics.

Chapter 1 explains the basic concepts and gives the overview of printing graphics. It also discusses the motivation that leads to this research.

Chapter 2 discusses the Bézier curve, its place in computer graphics and the previous work done on this topic and their drawback or limitations.

Chapter 3 states the hypothesis and discusses their reasonability with respect to the motivation of the research and the previous work done.

Chapter 4 contains the theoretical developments; it explains the analytical work done and the approximate solution of the problem, and discusses the approach features and limitation. It also contains the flow chart for the derived algorithm.

Chapter 5 first defines the practical test dataset used, then it explains the spatial and temporal performance improvement of parabolic approximation (the devised algorithm) versus the commonly used recursive subdivision algorithm. It also explains how can we tune the algorithm to get the best spatial performance.

Chapter 6 summarizes accomplishments of this research. It also lists some the questions that remained unanswered and may be good topics for future researches.

# TABLE OF CONTENTS

	Page
<b>ACKNOWLEDGEMENTS</b> .....	<b>i</b>
<b>PREFACE</b> .....	<b>iii</b>
<b>TABLE OF CONTENTS</b> .....	<b>iv</b>
<b>LIST OF FIGURES</b> .....	<b>vi</b>
<b>ABSTRACT</b> .....	<b>vii</b>
<b><i>Chapter 1</i> INTRODUCTION</b> .....	<b>1</b>
1.1 Computer graphics.....	1
1.1.1 Printing Graphics.....	2
1.2 Some useful definitions and terminology .....	3
1.3 Motivation for reducing number of primitives in the Display list.....	4
1.3.1 Discussion .....	5
<b><i>Chapter 2</i> BACKGROUND</b> .....	<b>7</b>
2.1 Bézier curves.....	7
2.2 Rendering Bézier curves.....	10
2.2.1 Bézier curve subdivision.....	11
2.2.2 Previous Work.....	12
2.2.2.1 Recursive Subdivision .....	12
2.2.2.2 Brute-Force Rendering.....	14
2.2.2.3 Forward differencing .....	14
2.2.2.4 Hybrid Approaches .....	15
<b><i>Chapter 3</i> HYPOTHESIS</b> .....	<b>16</b>
3.1 Hypothesis 1: .....	16
3.2 Hypothesis 2: .....	16
3.3 Discussion.....	16

<b>Chapter 4 THEORETICAL DEVELOPMENT .....</b>	<b>18</b>
4.1 Initial Attempt—Analytical Solution.....	19
4.2 Visual testing tool .....	20
4.3 Approximate Solution—Parabolic Approximation (PA).....	21
4.3.1 Parabolic approximation (PA) features and limitations .....	25
4.3.1.1 Reduced flatness for calculation .....	26
4.3.2 Inflection points.....	27
4.3.3 PA Algorithm description .....	29
<b>Chapter 5 EMPIRICAL PERFORMANCE ANALYSIS .....</b>	<b>32</b>
5.1 Performance Data Set .....	32
5.2 Spatial Performance.....	33
5.2.1 Spatial Performance versus Reduced Flatness .....	33
5.2.2 Spatial Performance Vs Flatness.....	36
5.3 Temporal Performance.....	37
5.3.1 Temporal Performance versus Reduced Flatness.....	38
5.3.2 Temporal Performance versus Flatness.....	39
5.4 Discussion.....	40
<b>Chapter 6 CONCLUSION .....</b>	<b>41</b>
6.1 Summary of current work .....	41
6.2 Future work.....	42
<b>REFERENCES.....</b>	<b>43</b>
<b>APPENDIX A Analytical Solution Work .....</b>	<b>45</b>
<b>APPENDIX B Calculating Inflection Points.....</b>	<b>48</b>
<b>APPENDIX C Code .....</b>	<b>50</b>
<b>VITA.....</b>	<b>55</b>

## LIST OF FIGURES

	Page
Figure 1. Example Primitives.....	2
Figure 2. Printing Graphics.....	2
Figure 3. Example polyline.....	3
Figure 4. Curve Approximation.....	4
Figure 5. Object Approximation and Trapezoidation.....	4
Figure 6. Example Bézier curves.....	7
Figure 7. Cubic Bézier Segment.....	9
Figure 8. Maximum Transverse Deviation $d_{\perp}$ .....	10
Figure 9. Subdivision of Curve $Q(t)$ .....	12
Figure 10. Recursive Subdivision Algorithm.....	13
Figure 11. Approximating Bézier Segment.....	19
Figure 12. Visual Test bed.....	20
Figure 13. Flattened Bézier curve segment.....	21
Figure 14. Parabolic approximation.....	22
Figure 15. Reduced Flatness.....	27
Figure 16. High Level PA Algorithm flow.....	29
Figure 17. Parabolic Approximation (PA) Algorithm Flow Chart.....	31
Figure 18. Average PA performance improvement against reduced flatness.....	34
Figure 19. PA performance improvement against reduced flatness.....	35
Figure 20. Spatial Performance of different flatness.....	36
Figure 21. Spatial Performance – RS v PS.....	37
Figure 22. Temporal Performance – PA improvement v reduced flatness.....	38
Figure 23. Temporal Performance – PA and RS running times versus flatness.....	39
Figure 24. Subdivision of Curve $P(t)$ .....	45
Figure 25. Bézier subsegment $S_p$ .....	46
Figure 26. Inflection point on Bézier curve.....	48

## **ABSTRACT**

Ahmad, Athar Luqman, M. S., University of South Alabama, May 2001, Approximation of a Bézier Curve with a Minimal Number of Line Segments, Chair of Committee: Dr. Thomas F. Hain.

Rendering a Bézier curve segment usually involves approximating the curve by a polyline. A well-known technique called recursive subdivision generates such an approximating polyline by subdividing the Bézier curve segment into two subsegments. Each subsegment is recursively subdivided until the maximum deviation of the subsegment from its chord is less than a predefined constant, the flatness. This ultimate subsegment can then be replaced by its chord. However, in this technique, subdividing curves slightly above the flatness threshold yields two subsegments whose maximum deviation may be much less than the flatness. The average deviation of all resultant subcurves will generally be less than the required flatness, implying that a greater number of approximating line segments than necessary will be generated. In the current research, the Bézier curve is subdivided in such a way that all ultimate subsegments are as long as possible, thus generating fewer polyline components. We have devised approximations and/or heuristics to ensure that the calculation overhead is acceptable.

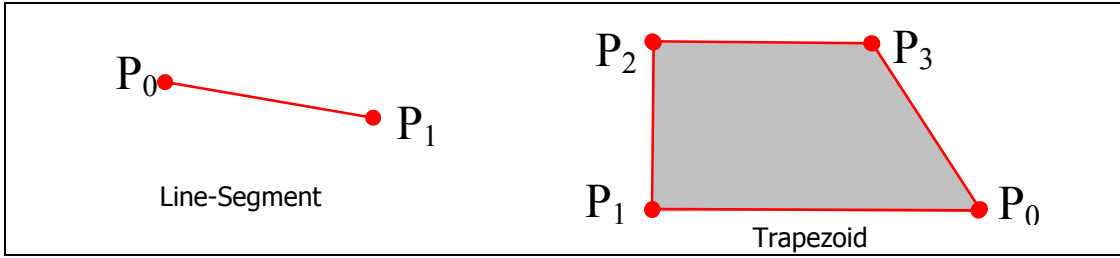
# Chapter 1

## INTRODUCTION

### 1.1 Computer graphics

Two-dimensional computer graphics drawing deals with conversion of high-level geometric objects—curves of various type thickness and style, areas with defined outline and fills/patterns, fonts, and bitmaps—to images. These images are finally rendered as pixels on various devices such as a monitor, or a printed page (produced by a printer). The pixels on the rendering device are normally addressed as a two-dimensional coordinate system known as a *raster*. The size of the raster in pixels is known as its *resolution*.

The high-level collection of objects, such as graphical objects and text fonts that are represented by graphical objects (excluding bitmaps), is typically converted to, and approximated by, a list of lower-level geometric primitives, namely *line-segments* (to approximate curves,) and *horizontally-aligned trapezoids* (to approximate areas.) This list is often called a *display list* [7]. The process of approximating the curve by line-segments is called the “flattening the curve.” The original image can thereby be stored in a (lossy) compressed form that can nevertheless be rapidly rendered. The degree of allowable approximation is dependent on the display resolution. Figure 1 shows an example of each primitive.

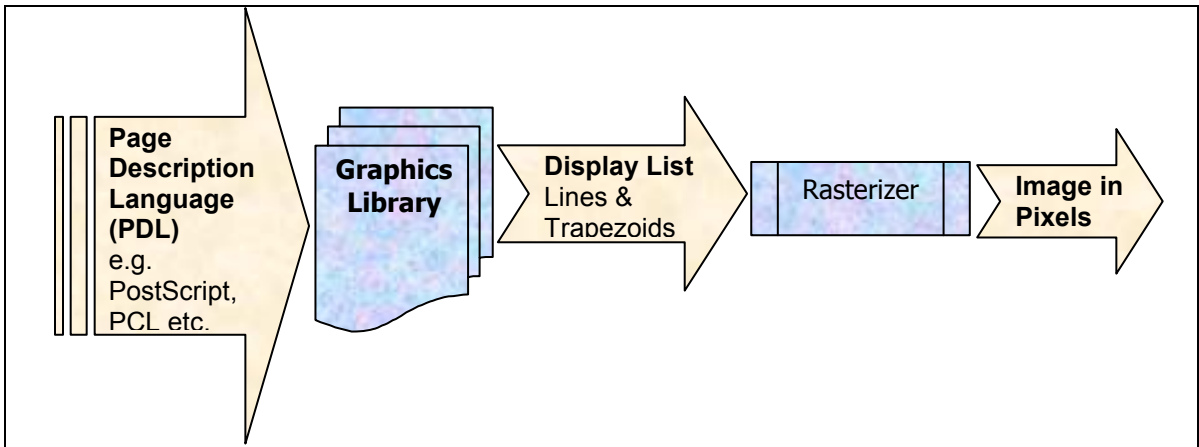


**Figure 1. Example Primitives**

### 1.1.1 Printer Graphics

Printer graphics deals with printing different high-level objects, such as images, vector graphical, and text objects, known as *page objects*, to a printing device. This process is described by Figure 2.

The printer gets the description of these objects in some predefined languages called *page description language (PDL)*, e.g. PostScript, PCL and PCL-XL. This description is passed to a graphics library which converts them to the primitives for the display list. The display list is then converted to the final image represented by pixels. This image can then be used by the device hardware to create the hard copy.



**Figure 2. Printing Graphics**

The intermediate step of converting the objects to display list is necessary because often we have to store the image in a compressed format for features like multiple copies, collation and duplexing (printing on both sides of the media). If we store the page objects

in PDL format then we will have to again spend the considerable time to convert it to image. On the other hand, storing them in image format needs relatively large amount of memory. Therefore, storing other than the display list can significantly reduce performance.

## 1.2 Some useful definitions and terminology

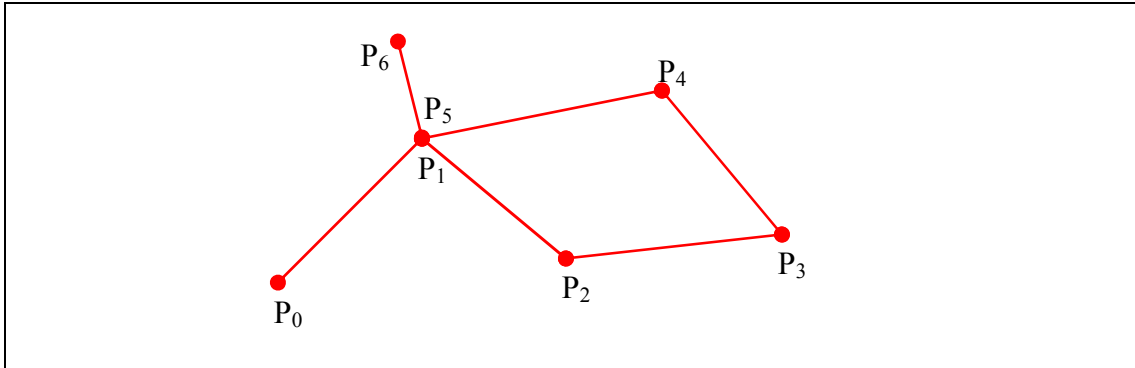
We now present some formal definitions and terminology useful in the current context.

Definition: A *polyline* is specified by a sequence of  $n \geq 2$  points

$(\mathbf{p}_0, \mathbf{p}_1, \mathbf{p}_2, \dots, \mathbf{p}_{n-2}, \mathbf{p}_{n-1})$ , such that adjacent points are non-coincident, and is composed of the sequence of (joined) line segments

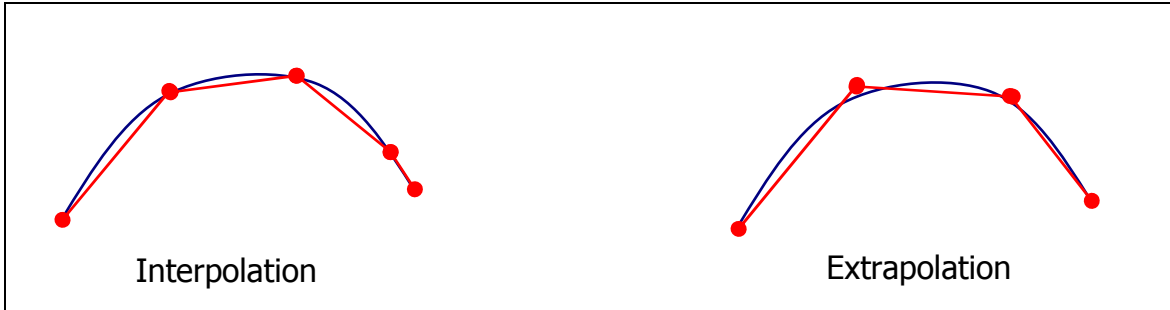
$(\mathbf{p}_0, \mathbf{p}_1), (\mathbf{p}_1, \mathbf{p}_2), \dots, (\mathbf{p}_{n-2}, \mathbf{p}_{n-1})$ . The points  $\mathbf{p}_0$  and  $\mathbf{p}_{n-1}$  are the polyline end-points, and the inner points, if they exist, are  $\mathbf{p}_1, \mathbf{p}_2, \dots, \mathbf{p}_{n-2}$ .

Figure 3 shows a polyline defined by 7 points  $(\mathbf{P}_0, \dots, \mathbf{P}_6)$ , where  $\mathbf{P}_1$  and  $\mathbf{P}_5$  are coincident.



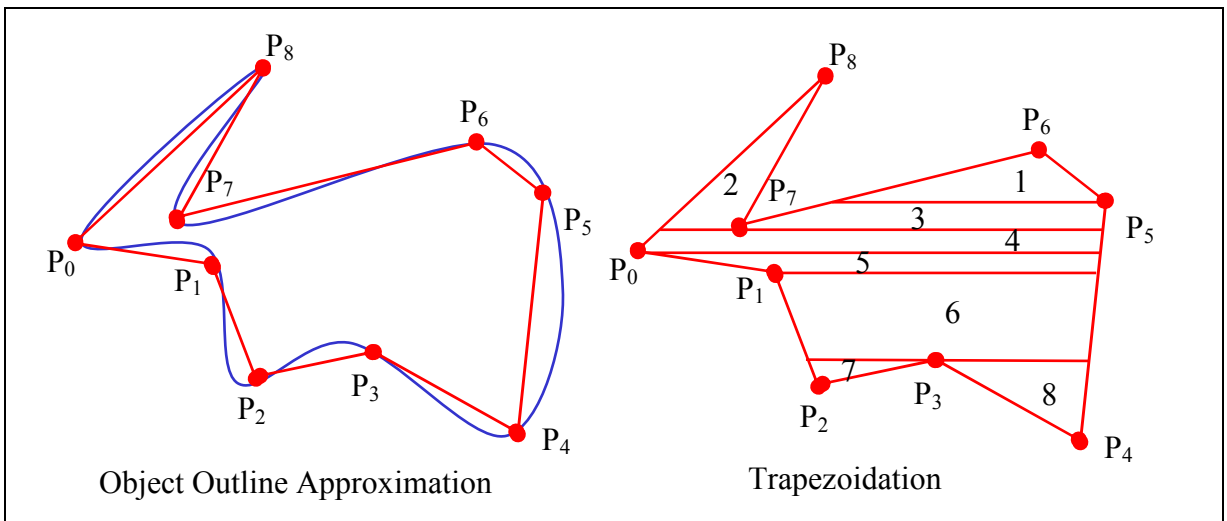
**Figure 3. Example polyline**

Curves are normally approximated by polylines, and we will say that a polyline *interpolates* a curve if the polyline end-points are coincident with curve end-points, and all the inner points of the polyline (if any exist) lie on the curve. If the inner points are not on the curve—but presumably close—we will say that the polyline *extrapolates* the curve. This research focuses on generating interpolating polylines. Figure 4 illustrates the concept.



**Figure 4. Curve Approximation**

On the other hand, areas are modeled by a collection of trapezoids. First we approximate the object outline by polyline and then the inner areas are resolved into a disjoint collection of trapezoids by using a technique called *trapezoidation*. In this technique we draw horizontal lines through each point on the polyline approximating the curve. Doing this divides the inner area of the curve into trapezoids. We do this because we know how to rapidly render trapezoids. Figure 5 shows an example of trapezoidation.



**Figure 5. Object Approximation and Trapezoidation**

### 1.3 Motivation for reducing number of primitives in the Display list

As explained in Section 1.1.1, we have to store the display list until it is converted to its pixel format when needed. The size of the display list is dependent on the number of

primitives in it; the smaller the number of line-segments and trapezoids in the display list the smaller will be its size.

Additionally, a smaller number of primitives will take less rendering time (which includes the conversion to the rasterized image object.) Each primitive requires a new and potentially complex initialization, after which the loop generating the pixels is relatively fast.

Therefore, reducing the size of the display list is beneficial from the point of view of both spatial and temporal performance. Thus, there is a strong motivation to resolve a drawing into the smallest number of primitives consistent with the resolution requirement.

The number of primitives can be reduced in following two major ways:

- *Curves*: Since curves are approximated by polyline, reducing the number of points in the polyline will decrease number of line-segments and hence the size of the display list.
- *Areas with curved boundaries*: Areas whose outlines are curves, are also important. The number of trapezoids needed to render an area object is proportional to the number of segments required to approximate (flatten) its outline curve. Typically, a polygon with  $n$  points is covered into  $n-1$  trapezoids. Thus, reducing the size of the outline approximation will reduce the number of trapezoids, and hence display list.

### **1.3.1 Discussion**

These benefits are somewhat theoretical in nature. In practice it is very likely that we have a tradeoff between display list generation speed, display list storage space, and image rendering speed, all of which is implementation as well as application dependent. Reducing the number of points in a polyline would decrease the display list or trapezoids needed but may increase the calculation overhead in approximation process. The greatest advantage is to be gained when the same image needs to be rendered several times with a

single generation of the display list (e.g., printing multiple collated copies of multi-page documents). In this case the display list generation can usually be amortized over multiple renderings (unless display list memory is limited, or image complexity is too great.)

## Chapter 2

### BACKGROUND

#### 2.1 Bézier curves

Other than conics, cubic Bézier curves are the most commonly used curves in computer graphics. Bézier curves offer a way to efficiently represent smooth curves as well as curved surfaces (in 3-D graphics). One can draw, with arbitrary precision, almost any type of graphical object using this type of curve. Moreover, a relatively small number of cubic Bézier curve segments are needed to approximate a given higher degree curve. A cubic Bézier curve segment is efficiently specified using only four control points. The first and the last points are interpolated by the curve, while the other two are approximated by it. Figure 6 illustrates examples of Bézier curves.

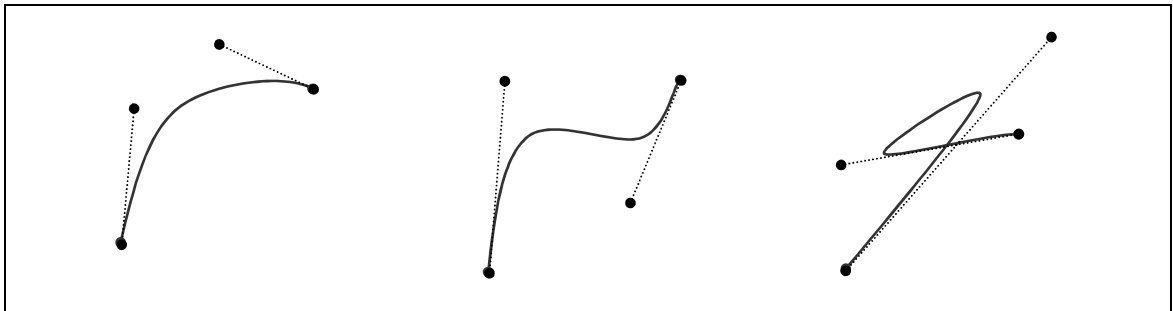


Figure 6. Example Bézier curves

**Definition:** Points on an *order- $n$  Bézier curve segment*<sup>1</sup>,  $\mathbf{Q}(t)$ , can be expressed parametrically [1][4][5] as

$$\mathbf{Q}(t) = \sum_{i=0}^n \mathbf{P}_i B_{n,i}(t)$$

where

1.  $0 \leq t \leq 1$
2.  $\mathbf{P}_i$  are control points
3.  $B_{n,i}(t) = \binom{n}{i} t^i (1-t)^{n-i}$  are known as *Bernstein polynomials, or blending functions*.

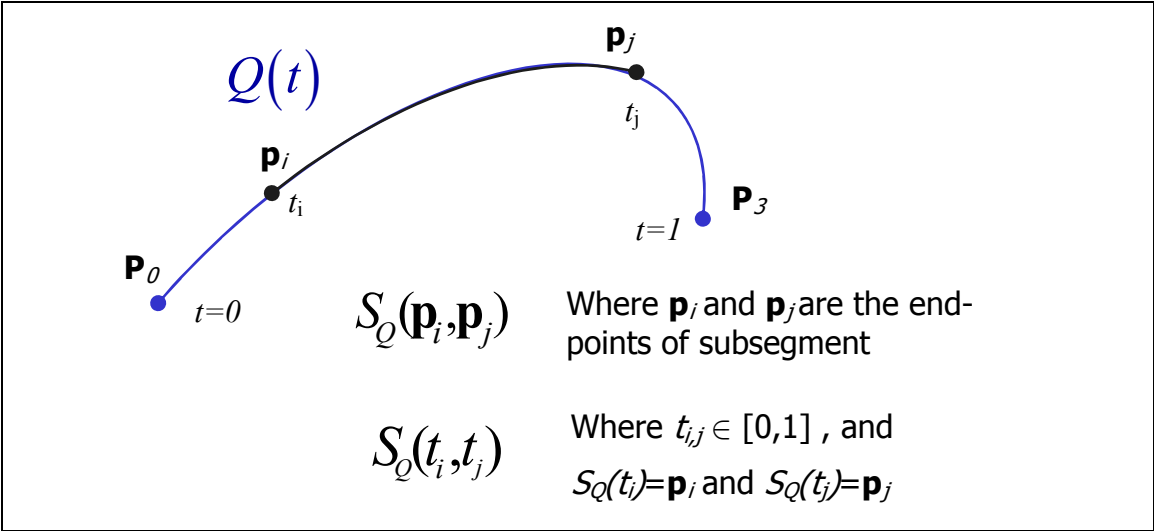
‘Parametrically’ means that by substituting different values of  $t$  in the range  $[0,1]$  we get different points on the curve segment. If we substitute 0 for  $t$  we get first control point,  $\mathbf{P}_0$ , and if we substitute 1 we get the last control point,  $\mathbf{P}_n$ . It is clear from the definition that the equation is a polynomial of order  $n$ , and that  $n+1$  characteristic (control) points are needed to specify an order- $n$  Bézier curve. Higher order Bézier curves may have more oscillations than lower order ones, and so are not suitable for representing relatively simple smooth curves. On the other hand, 1<sup>st</sup> and 2<sup>nd</sup> order curves are too simple to compose complex curves since, in general, they provide too few degrees of freedom to allow smooth joints. Most commonly, cubic (3<sup>rd</sup> order) curves are used in simple drawing applications. In this research, when we refer to Bézier curves, we will mean two-dimensional cubic curves unless otherwise stated.

**Definition:** A cubic Bézier curve *segment* is specified by four control points,  $P_{0...3}$ . The outer control points,  $P_0$  and  $P_3$ , are the curve segment end-points. The inner control points,  $P_1$  and  $P_2$ , control the tangential directions of the curve at  $P_0$  and  $P_3$  respectively, and the lengths of the  $\overline{P_0P_1}$  and  $\overline{P_2P_3}$  line segments control to what extent  $P_1$  and  $P_2$  “pull” the curve into their direction.

---

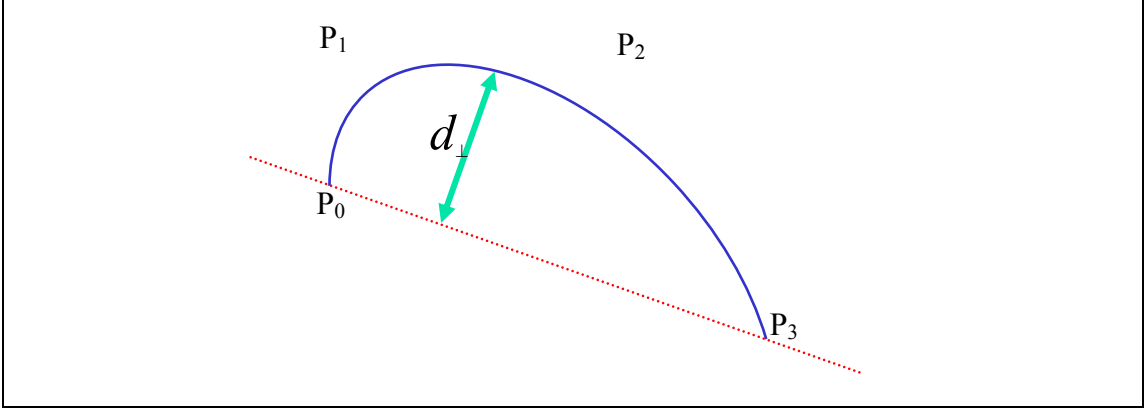
<sup>1</sup> A Bézier curve segment is distinguished from a Bézier curve in the same way that a line segment is distinguished from a line; it is the region of the curve between defined endpoints.

**Definition:** A subsegment of a Bézier curve  $Q$  is represented as  $S_Q(\mathbf{p}_i, \mathbf{p}_j)$ , where  $\mathbf{p}_i$  and  $\mathbf{p}_j$ , are the end-points of the subsegment (and are on  $Q$ ). We may also define a subsegment by the corresponding values of the parameter  $t$  at the endpoints,  $S_Q(t_i, t_j)$ , with the value of  $t$  being 0 and 1 on the endpoints of the (host) Bézier curve (See Figure 7.)



**Figure 7. Cubic Bézier Segment**

**Definition:** For a cubic Bézier curve (sub)segment  $Q$  the *maximum transverse deviation*,  $d_{\perp}(Q)$ , is defined as follows: if  $\mathbf{P}_0$  and  $\mathbf{P}_3$  are non-coincident end-points then it is the maximum perpendicular distance of any point on  $Q$  from the line passing through  $\mathbf{P}_0$  and  $\mathbf{P}_3$  otherwise it is the maximum distance of any point on  $Q$  from  $\mathbf{P}_0$ . (See Figure 8.)



**Figure 8. Maximum Transverse Deviation  $d_{\perp}$**

The following definition will become useful in the following discussion.

**Definition:** A *point sequence* on a Bézier curve described by function  $\mathbf{Q}(t)$  is denoted as either  $\mathbf{P}_Q(t_0, t_1, \dots, t_{n-1})$ , or  $\mathbf{P}_Q(\mathbf{p}_0, \mathbf{p}_1, \dots, \mathbf{p}_{n-1})$  such that  $\mathbf{p}_i = \mathbf{Q}(t_i)$ ,  $0 \leq i \leq n-1$ , is a strictly monotonically increasing sequence of parametric values  $t_i$ , i.e.,  $0 = t_0 < t_1 < \dots < t_{n-1} = 1$ . For a given  $n$ , there is actually an infinite set of such point sequences.

## 2.2 Rendering Bézier curves

Rendering a Bézier curve segment usually involves first transforming the curve to an approximating polyline. A polyline interpolates a Bézier curve  $\mathbf{Q}$  if the sequence of points that defines it is a point sequence of  $\mathbf{Q}, P_Q$ . The greatest transverse deviation of the polyline from the curve is given by

$$\max_{0 \leq i < n-1} \left( d_{\perp} \left( S_Q(\mathbf{p}_i, \mathbf{p}_{i+1}) \right) \right) \quad (1.1)$$

Where  $S_Q$  is a Bézier segment from  $\mathbf{p}_i$  to  $\mathbf{p}_{i+1}$ . Generally, when we are trying to render a Bézier curve by using an interpolating polyline, we wish to put an upper bound on this value (for example, one pixel width), since it is related to the accuracy with which the polyline renders the Bézier curve.

**Definition:** the *flatness*  $f$  of curve rendering is a specified upper bound on the greatest transverse deviation of a polyline from a curve. That is, for all acceptable polyline renderings of a Bézier curve  $Q$  we have

$$\max_{0 \leq i < n-1} \left( d_{\perp} \left( S_Q(\mathbf{p}_i, \mathbf{p}_{i+1}) \right) \right) \leq f \quad (1.2)$$

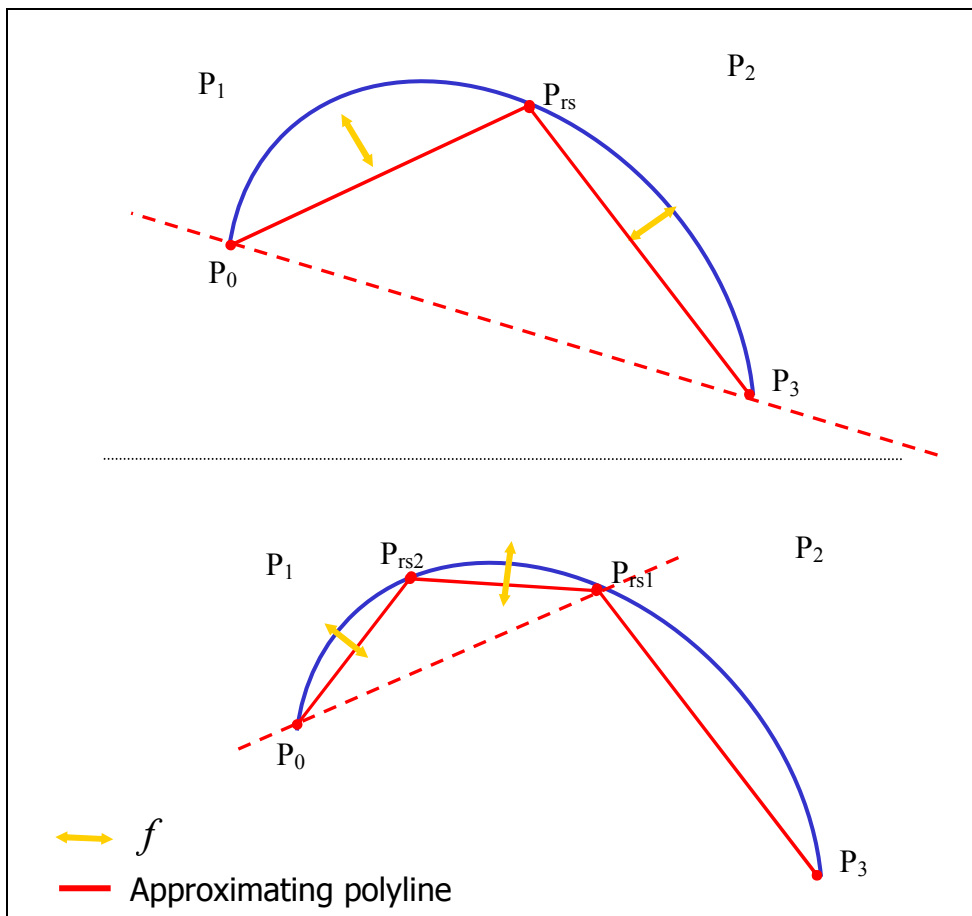
### 2.2.1 Bézier curve subdivision

It is the property of Bézier curves that if one divides a curve segment into smaller subsegments the resultant subsegments are also Bézier curves. Division of the Bézier curve segment at some particular  $t$  value in two smaller Bézier subsegment is called Bézier curve subdivision.

Subdividing the curve is a geometric technique developed by de Casteljau [6]. Mathematically finding the control points of the two subsegments is simple and involves only twelve additions and 6 multiplications [7]. The geometric interpretation and equations used for the subdivision calculation is shown in Figure 9.  $P_0, P_0', P_0''$ , and  $P_0'''$  are the control points of the first Bézier curve, while  $P_0''', P_1'', P_2'$ , and  $P_3$  are the control points of the second Bézier curve.



smaller subsegments. This dividing process is continued recursively until the each of the subcurves has the flatness under the tolerance value. Straight-line segments can then approximate these subcurves. Figure 10 shows that during first subdivision we have maximum transverse deviation,  $d_{\perp}$ , of one of the curves less than the required flatness,  $f$ , while other curve's  $d_{\perp}$  does not satisfy the required  $f$ . So we divide the second subcurve again to get the required results. Final approximating polyline is given by polyline  $P_0, P_{rs2}, P_{rs1}, P_3$ .



**Figure 10. Recursive Subdivision Algorithm**

Hain[9] has shown an efficient way to calculate a much more accurate (but still conservative) method to estimate the minimum transverse deviation. This causes, on average, earlier termination of the recursion, and consequently, on average, an interpolating polyline with fewer points. It should be noted that, when a segment whose (estimated) maximum transverse deviation is barely above the flatness is split, the two resulting subsegments will have minimum transverse deviations well below the flatness. This will result in a polyline with a cardinality that is not minimal.

### **2.2.2.2 Brute-Force Rendering**

Brute-Force is an iterative process in which we find the point on the curve by plugging in the successive values of  $t$  using Horner's rule[7]. The cost for each point calculation is 9 multiplies and 10 additions. This is very expensive and also do not take account of the curvature of the cure. This technique is rarely used in the industry.

### **2.2.2.3 Forward differencing**

Another technique used for approximating Bézier curves by polyline is known as forward differencing [7][10]. In this technique, we divide  $t [0,1]$ , into  $n$  equally spaced intervals,  $\delta$ , such that  $n\delta = 1$ , and then find the function value at the endpoints of each interval. Since cubic Bézier curve is third degree polynomial and its third derivative is always a constant, we can compute the point at next  $\delta$  by finding up to third forward differences. So finding first three points involve a little expensive calculation, but the rest of the points can be computed only with a few additions. The resultant set of coordinates defines the curve. Below are first three differences of the initial point i.e., at  $t=0$ , for cubic Bézier curve given by  $P(t) = At^3 + Bt^2 + Ct + D$ .

$$P(0) = p_0 = D$$

$$\Delta p_0 = A\delta^3 + B\delta^2 + C\delta$$

$$\Delta^2 p_0 = 6A\delta^3 + 2B\delta^2$$

$$\Delta^3 p_0 = 6A\delta^3$$

After the initial set of computation rest of the points and their forward differences are calculated using the following formulas.

$$p_n = p_{n-1} + \Delta p_{n-1}$$

$$\Delta p_n = \Delta p_{n-1} + \Delta^2 p_{n-1}$$

$$\Delta^2 p_n = \Delta^2 p_{n-1} + \Delta^3 p_{n-1}$$

The biggest drawback of this approach is that it always generates a fixed number of line segments depending upon the number of division of the parametric value, and does not take into account the curve's curvature. Thus, if we keep the criterion of trying to generate the fewest segments, this technique is definitely not an appropriate approach.

#### ***2.2.2.4 Hybrid Approaches***

However, a hybrid of the recursive subdivision and forward differencing may yield better results. One of the hybrid approaches is known as adaptive forward differencing [11][14][13]. This technique uses features from both recursive subdivision and adaptive forward differencing. The main strategy here is forward differencing, but it uses computationally efficient method to adjust the step size by factors of two. This still does not lead to an optimal number of approximating segments, which is the goal of the current research.

## Chapter 3

### HYPOTHESIS

The benefits for reducing the number of primitives in the display list (Section 1.3) give us the motivation for developing an algorithm that will reduce the number of line-segments generated in an approximation (flattening) of the Bézier curve segment. Classifying the properties of the algorithm into spatial and temporal, we have formulated two hypotheses.

#### **Hypothesis 1:**

An algorithm can be devised that generates a polyline approximation (within a given flatness) of any Bézier curve that has no more line-segments, but on average 15% fewer line-segments, than approximations generated by recursive subdivision (using Hain's stopping criteria. [9])

#### **Hypothesis 2:**

The algorithm will incur no more than a 100% computational penalty (intuitively estimated), under implementation on a general-purpose single processor computer.

### **3.1 Discussion**

It is clear that, of all algorithms flattening a Bézier curve to within the flatness constraint, that one with the greatest average transverse deviation over all segments will have the fewest segments. The challenge, taken up in this research, is to develop an efficient algorithm to find a reduced set of points satisfying Equation 1.2. This will occur if the points are chosen such that one of the following conditions holds:

$$d_{\perp}(S_Q(\mathbf{p}_i, \mathbf{p}_{i+1})) = f, \quad 0 \leq i < n - 2$$

$$d_{\perp}(S_Q(\mathbf{p}_i, \mathbf{p}_{i+1})) = f, \quad 1 \leq i < n-1$$

That is, we start from one or the other end-point of the Bézier curve, and choose the next point  $p_{next}$  as far along the curve as possible such that  $d_{\perp}(S_Q(\mathbf{p}, \mathbf{p}_{next})) = f$  where  $\mathbf{p}$  is the previous point. The last Bézier subsegment will have the property  $d_{\perp}(S_Q(\mathbf{p}, \mathbf{p}_{last})) \leq f$  where  $\mathbf{p}$  is the penultimate point, and  $\mathbf{p}_{last}$  is the last point of the polyline.

This approach does not guarantee the fewest possible line segments because it only looks at the interpolating polynomial approach. In some cases extrapolating the curve can cover larger part of the curve. Figure 4 shows that when same curve is extrapolated produce one less line-segment as an approximation. However, it would generate fewer segments than the most commonly used recursive subdivision algorithm, without incurring excessive computational overhead. We conjecture that optimality (within the constraint that segment end-points fall on the curve—the interpolation case) will be achieved starting at either end.

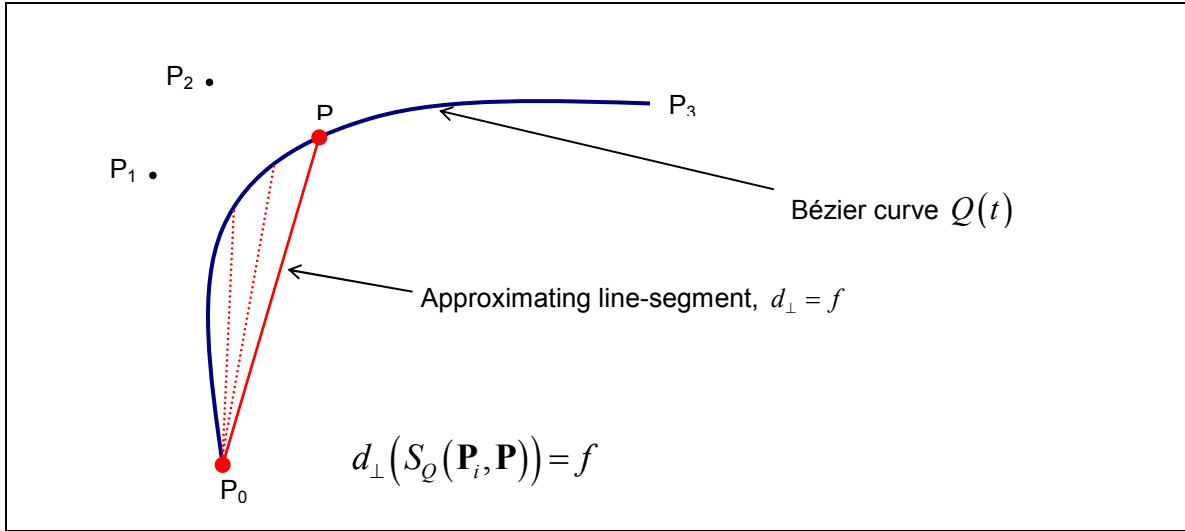
It should also be noted that recursive subdivision using Hain's stopping criteria [9], rather than the common stopping criterion, already reduces the number of line segments by 26% from the most commonly used stopping criteria, which represents a close to optimal improvement for recursive subdivision. Our technique is compared to the optimal recursive subdivision algorithm used as a baseline, and is shown to significantly reduce the number of segments even further.

## Chapter 4

### THEORETICAL DEVELOPMENT

Say that control points  $\mathbf{P}_0, \mathbf{P}_1, \mathbf{P}_2$  and  $\mathbf{P}_3$  specify a cubic Bézier curve  $\mathbf{Q}(t)$ . A line-segment  $\overline{\mathbf{P}_0\mathbf{P}}$ , where  $\mathbf{P}$  is a point on the curve, is an approximation of the Bézier subsegment  $S_Q(\mathbf{P}_0, \mathbf{P})$ . We need to find a point  $\mathbf{P}$  on the curve so that the Bézier subsegment has its maximum transverse deviation equal to the maximum tolerance (flatness),  $f$ , i.e.,  $d_{\perp}(S_Q(\mathbf{P}_0, \mathbf{P})) = f$ .

The end-point  $\mathbf{P}$  of the line segment would then be taken as the starting point of the remaining curve for which we iterate to find the next longest possible Bézier subsegment having maximum deviation equal to flatness. This process would be iterated until point  $\mathbf{P}_3$  is reached.

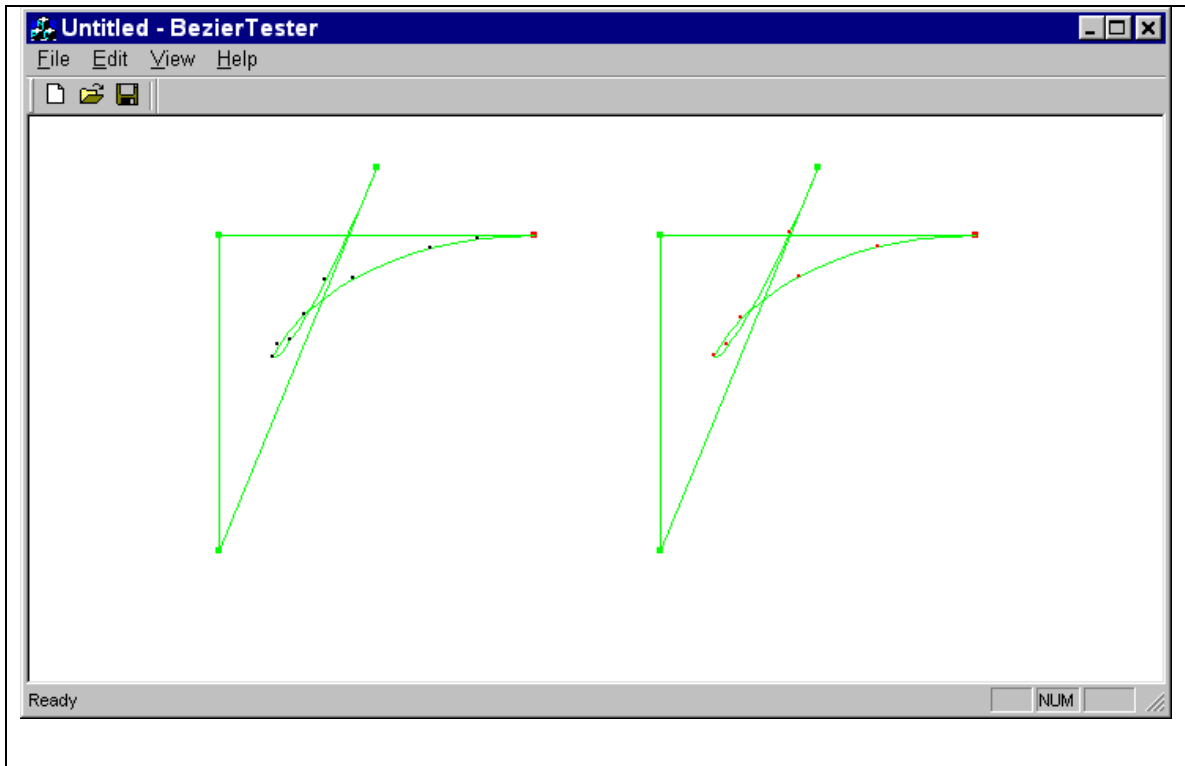


**Figure 11. Approximating Bézier Segment**

#### 4.1 Initial Attempt—Analytical Solution

In the analytical part of the research we tried to find the mathematical solution of the problem. In other words, we tried to derive a formula that would give us the  $t$  value such that if we divide the cubic Bézier curve segment at that value the first Bézier subsegment has its maximum deviation equal to the required flatness.

After considerable scrutiny and work, a partial solution was derived, but it was determined that an analytical solution would be expensive and most certainly will not be feasible for any practical algorithm. The details of this work are given in the Appendix A.



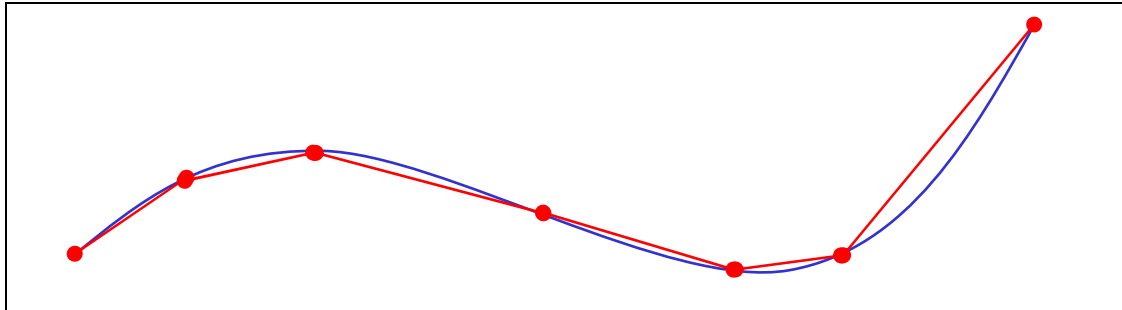
**Figure 12. Visual Test bed**

## **4.2 Visual testing tool**

During the development of the algorithm, an interactive visual test bed program was created that shows the Bézier curve and the points of the approximating polyline. Figure 12, is a screen shot of the Windows application, where the points on the left segment are approximated using recursive subdivision (RS) and the points on the right segment are approximated using parabolic approximation (PA – current research as explained in section 4.3). The control points of one of the Bézier curves could be interactively dragged to arbitrary positions, and the effect of changing the curve could be visually observed for both algorithms. This visual tester not only gave us a tool to test the flattening algorithm, but also played an important role in making us understand different Bézier shapes, and their effect on the correctness constraints of the algorithms.

### 4.3 Approximate Solution—Parabolic Approximation (PA)

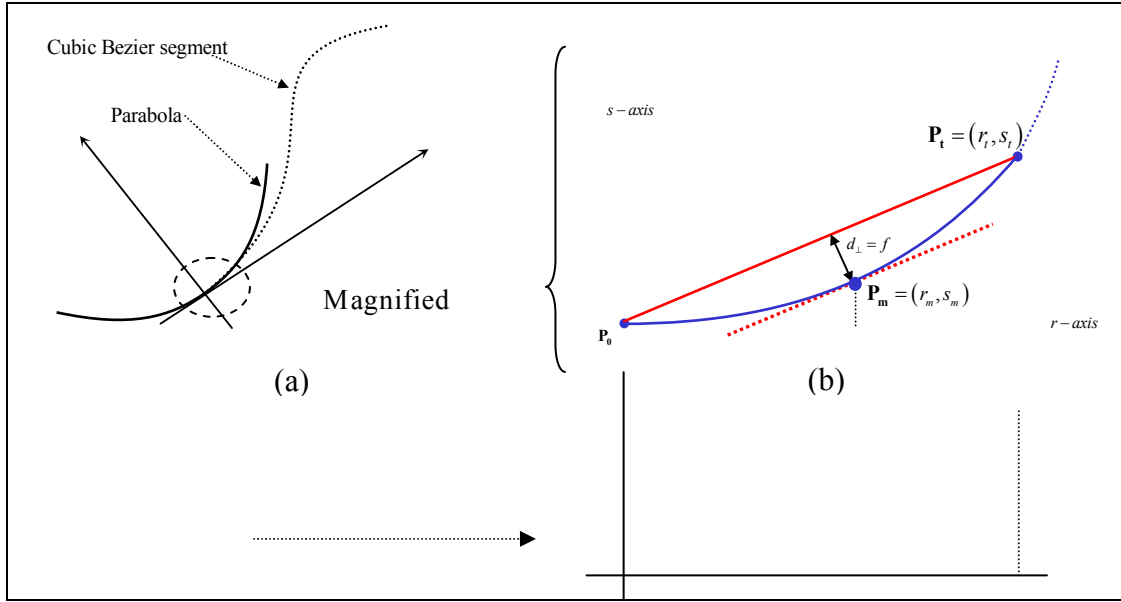
With the help of this tool we noticed that after the Bézier curve segment is flattened the Bézier subsegments that will be replaced by line-segments are a reasonable approximation to a parabola, as can be seen in the Figure 13.



**Figure 13. Flattened Bézier curve segment**

This idea led us to the idea to approximate the cubic Bézier curve (sub)segment with a parabola. This reduced considerably the complicated algebra, and simplified the solution. Figure 14(a) demonstrates the situation where there is significant resemblance of the cubic Bézier curve subsegment and a parabola making it a candidate for an approximation.

We have defined a coordinate system such that its origin is at the first control point of the Bézier curve subsegment and one of its axes is tangential to the curve at that point. If we put a parabola such that it passes through the origin and is also tangential to the same axis as the Bézier curve, then the parabola and the cubic Bézier curve segment overlap one another, at least in the region around the origin.



**Figure 14. Parabolic approximation**

Figure 14(b) shows the magnified picture of the overlapping of the Bézier curve subsegment and its approximation parabola. The solid curve represent parabolic subsegment, being an approximation to the Bézier subsegment,. The dotted curve represents the rest of the cubic Bézier curve segment. We need to find the  $t$  value at point  $\mathbf{P}_t$  on the parabolic subsegment, such that the maximum transverse deviation,  $d_{\perp}$  is equal to the required flatness,  $f$ . This point approximates the corresponding point on the Bézier segment.

Consider a normalized coordinate system with  $s$ - and  $r$ -axes such that the origin is at  $\mathbf{P}_0$  and the  $r$ -axis is in the direction of  $\overline{\mathbf{P}_0\mathbf{P}_1}$ , where  $\mathbf{P}_0$  and  $\mathbf{P}_1$  are the first two control points of the Bézier segment, and the  $s$ -axis is orthogonal to this in the right-handed sense. An arbitrary point  $\mathbf{P}_c(x, y)$  in this new system can be transformed into  $\mathbf{P}_c(r, s)$ , using the new coordinates as follows:

$$r = \frac{(P_{0_y} - P_{c_y})(P_{0_y} - P_{1_y}) - (P_{0_x} - P_{c_x})(P_{1_x} - P_{0_x})}{L^2}$$

$$s = \frac{(P_{0_y} - P_{c_y})(P_{1_x} - P_{0_x}) - (P_{0_x} - P_{c_x})(P_{1_y} - P_{0_y})}{L^2}$$

where,

$$L = \|\mathbf{P}_0\mathbf{P}_1\| = \sqrt{(P_{1_x} - P_{0_x})^2 + (P_{1_y} - P_{0_y})^2}$$

Let,  $\mathbf{P}_m = (r_m, s_m)$  be the point on the Bézier subsegment,  $\mathbf{Q}(\mathbf{P}_0, \mathbf{P}_t)$ , such the distance of point  $\mathbf{P}_m$  from the line  $\overline{\mathbf{P}_0\mathbf{P}_t}$  is equal to maximum transverse deviation,  $d_{\perp}$ . The goal of this setup is to find the value of  $t$  in terms of  $d_{\perp}$ . So we need to find some relation between point  $\mathbf{P}_m$  and  $\mathbf{P}_t$ . The parabola and its slope is defined as:

$$s = ar^2, \quad \frac{ds}{dr} = 2ar$$

Therefore the slope at  $\mathbf{P}_m$  is:

$$2ar_m$$

Since maximum transverse deviation occurs at  $\mathbf{P}_m$ , its slope is equal to the slope of the line-segment  $\overline{\mathbf{P}_0\mathbf{P}_t}$  and is given by:

$$\begin{aligned} \frac{ar_t^2}{r_t} &= 2ar_m \\ \Rightarrow r_m &= \frac{1}{2}r_t \end{aligned}$$

Similarly:

$$s_m = \frac{a}{4} r_i^2$$

The equation of the line  $\overline{\mathbf{P}_0\mathbf{P}_i}$  is given as:

$$\Rightarrow \frac{ar_i^2}{r_i} r - s = 0$$

Since we are in the normalized coordinate system we will have to divide it by the length of the line to get the equation in our system.

$$\frac{ar_i}{\sqrt{a^2 r_i^2 + 1}} r - \frac{s}{\sqrt{a^2 r_i^2 + 1}} = 0$$

The distance from the above line of the point  $\mathbf{P}_m$  can be calculated by substituting the point coordinates in the line equation. This distance is defines as maximum transverse deviation  $d_{\perp}$ . After simplifying for  $d_{\perp}$  and solving for  $r_i$  we get:

$$\begin{aligned} d_{\perp} &= \frac{ar_i}{\sqrt{a^2 r_i^2 + 1}} r_m - \frac{1}{\sqrt{a^2 r_i^2 + 1}} ar_m^2 \\ &= \frac{ar_i}{\sqrt{a^2 r_i^2 + 1}} \left( \frac{r_i}{2} \right) - \frac{1}{\sqrt{a^2 r_i^2 + 1}} \left( a \frac{r_i^2}{4} \right) \\ &= \frac{ar_i^2}{4\sqrt{a^2 r_i^2 + 1}} \\ &\Rightarrow r_i = 2\sqrt{a^2 r_i^2 + 1} \sqrt{\frac{d_{\perp}}{a}} \end{aligned}$$

Note that it is reasonable to assume that  $(ar_i)^2 \approx 0$  because the angle between the cord  $\overline{\mathbf{P}_0\mathbf{P}_i}$  and the Bézier curve subsegment is very small. Therefore  $r_i$  can be approximated as:

$$r_t \cong 2\sqrt{\frac{d_{\perp}}{a}}$$

To solve the value of the ‘ $a$ ’ we will look at the coefficient of the square power of the  $s$ -component of the cubic Bézier curve in its simplified polynomial form.

$$P_s(t) = P_{0_s} + (3P_{1_s} - 3P_{0_s})t + (3P_{2_s} - 6P_{1_s} + 3P_{0_s})t^2 + (P_{3_s} - 3P_{2_s} + 3P_{1_s} - P_{0_s})t^3$$

Since in the normalized system  $s$ -component of the first two control points were zero, the 2<sup>nd</sup> power coefficient is given as  $3P_{2_s}$ . Therefore,

$$r_t \cong 2\sqrt{\frac{d_{\perp}}{3P_{2_s}}}$$

Now we make another assumption that our  $r$ -component changes at a constant rate with comparison to the  $t$  value. Such that  $dr/dt = \text{constant}$ . Therefore,

$$\boxed{t \cong 2\sqrt{\frac{d_{\perp}}{3P_{2_s}}}}$$

#### 4.3.1 Parabolic approximation (PA) features and limitations

In PA we find the  $t$  values where the maximum transverse deviation,  $d_{\perp}$ , on the parabolic approximation is equal to flatness. Then we subdivide, as shown in section 2.2.1, the cubic Bézier curve segment at that value, the first part of the curve will give us the Bézier subsegment that could be replaced with its respective cord, and the rest of the Bézier subsegment is passed again through the same process. This process is repeated until the leftover Bézier subsegment also satisfies flatness.

One of the best features of the parabolic approximation (PA) is that the mathematical computation involved with respect to analytical solution is quite simple. More importantly, the PA solution leads us to a non-recursive algorithm rather than the

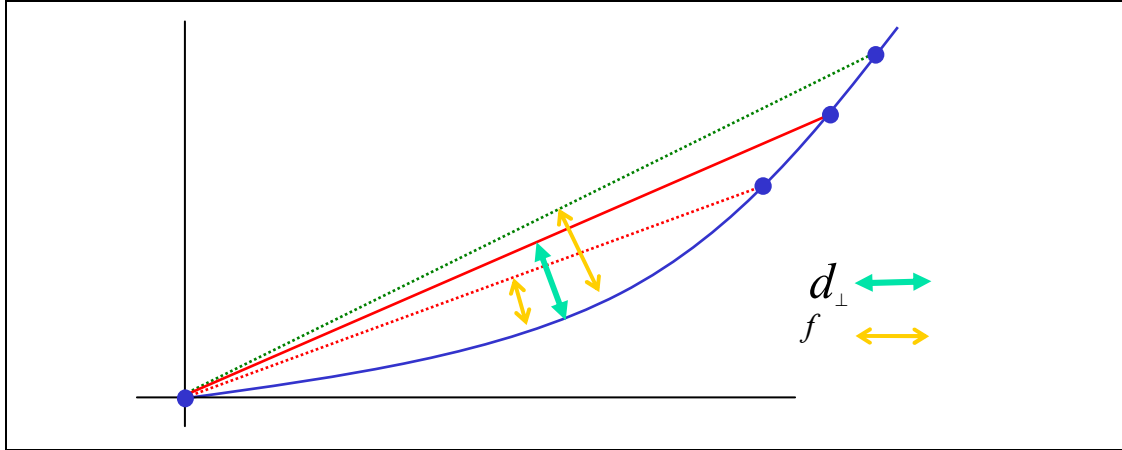
recursion required in recursive subdivision (RS). In PA, the flattening of the Bézier curve segment can be done using iteration. This is a big advantage since, for the majority of types of processors, recursion is slower than iteration.

Another important thing to notice is that in calculating the  $t$  value for subdivision, only the first 3 control points take part. We set our normalized coordinate system such that the new system  $r$ -axis is aligned with line passing through first two control points and then our solution has third control point in its calculation. It is independent of the fourth control point  $\mathbf{P}_3$ . This makes sense since we are approximating using a parabola, a second degree polynomial, and if we expand the cubic Bézier curve segment in polynomial form, the fourth control point is part of only the coefficient of the cube term.

In the equation above, the third control point represents its deviation from the new coordinate system  $r$ -axis. If this deviation is relatively small as compared to the distance between the first two control points then the calculated  $t$  value will have a higher probability of being erroneous. Moreover, if all of the control points are collinear then point  $\mathbf{P}_3$  in the equation will be zero hence leading to an incorrect value. On the other hand, if all the control points are at a fairly uniform distance from one another—a common situation for subsegments meeting the flatness requirement—a parabolic approximation is very good since it strongly resembles the Bézier subsegment.

#### ***4.3.1.1 Reduced flatness for calculation***

Since the PA uses a parabolic approximation, the calculated value of  $t$  may fall on either side of the “true” value on the Bézier curve. If the calculated value is a little more than the true value, the maximum transverse deviation,  $d_{\perp}$ , will not satisfy flatness. To lessen the chance that any of the Bézier subsegment’s deviation is greater than flatness, PA calculates the value of  $t$ , using flatness value a little less than the required flatness, called the *Reduced Flatness*,  $f_r$ .



**Figure 15. Reduced Flatness**

In the Figure 15 the solid line-segment is the approximation generated by PA. If the actual  $t$  value was a little larger than the approximated value as shown with dotted line-segment with a higher slope, then the deviation of the approximated Bézier subsegment will still satisfy the flatness. On the other hand, in cases where the deviation of the approximated Bézier subsegment is a little larger than the acceptable value, or the approximated  $t$  value is smaller than the actual value, as shown with a dotted line-segment whose slope is smaller the approximated line-segment, we will have to divide the Bézier subsegment at least one time to make sure that all the Bézier subsegments satisfy the flatness. If we use the reduced flatness to calculate the  $t$  value then the effect of error in approximation can be eliminated, or at least there will be a lower probability that we have to divide the approximated Bézier subsegment.

### 4.3.2 Inflection points

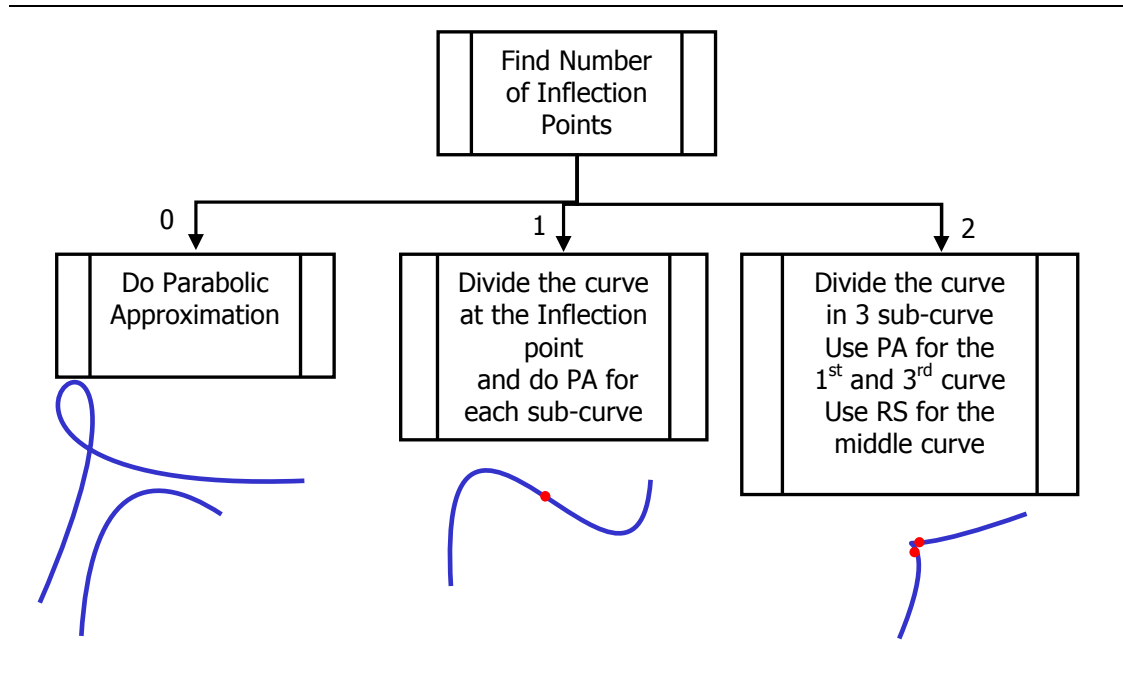
The parabolic approximation of cubic Bézier curve segment is dependent on the direction of the change of slope of the curve when moving on the curve away from the first control point. Since the approximation is quadratic polynomial (parabolic) it will only be an acceptable approximation in cases where the curve does not change the sign of the curvature. Therefore, inflection points, if present in the curve, must be considered.

If we divide the curve at inflection points then theoretically the resultant Bézier subsegments should not have any inflection point, hence favoring the parabolic approximation. Figure 16 gives us the high level of the algorithm flow, it filters out the cases of inflection point for PA to work efficiently.

Since cubic Bézier curve segment can have at most two inflection points, we have three cases. In the first case, where the curve does not have any inflection point, we can use the PA algorithm directly. This is the most common case as evidenced by the practical data set collected (section 5.1).

In the second case, where curve has one inflection point, we divide the curve at that inflection point and apply the approximation on the resultant Bézier subsegments separately. We must be careful when compiling the list of points for the polyline as the endpoint of the first Bézier subsegment and the start point of the second Bézier subsegment are the same.

The third and final case is where we have two inflection point. This is a very rare case (section 5.1) because in practice Bézier curves are used to represent relatively smooth curves or curved outlines. In this case, we divide the curve into three Bézier subsegments. First we subdivide into two Bézier subsegments at the  $t$  value which is smaller of the two inflection point's  $t$  values. This will insure that the second inflection point lies in the second Bézier subsegment. Then we find the new  $t$  value of the inflection point in the second Bézier subsegment before dividing it. Theoretically there should be one on inflection point left in the second curve. It was noted that the second (middle) Bézier subsegment in most cases was dumbbell in shape. Bézier curve segments such as these have second and third control points very close to one another, and PA is not suitable for them. It was empirically determined that it is best to use recursive subdivision to approximate the second Bézier subsegment. Fortunately, curves with two inflection points are rare, so the choice of the algorithm for the second Bézier subsegment does not effect the overall performance. In addition, second Bézier subsegments tend to be quite short.



**Figure 16. High Level PA Algorithm flow**

Appendix B discusses the details of finding the inflection points.

### 4.3.3 PA Algorithm description

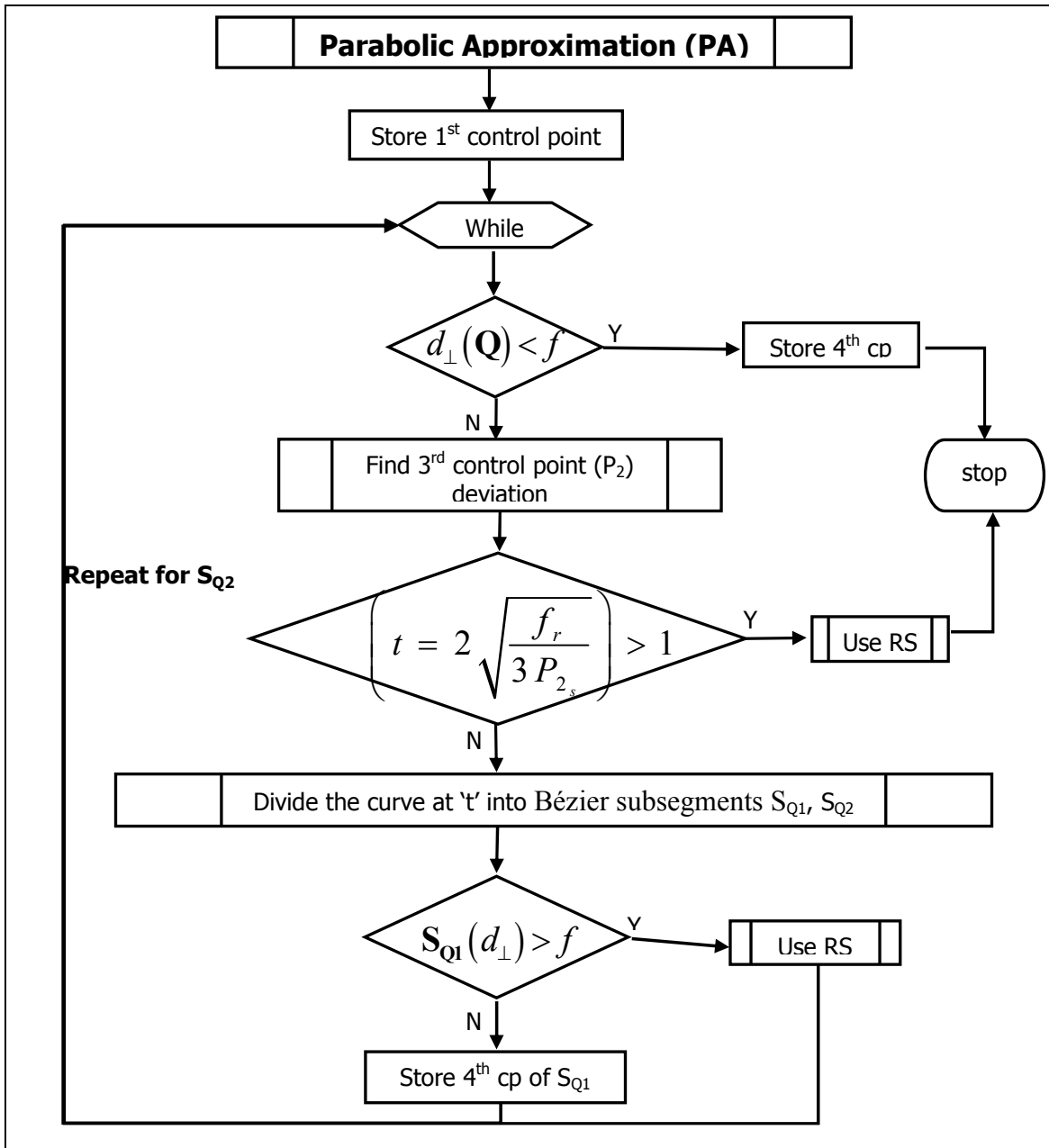
In the parabolic approximation algorithm, we move along the Bézier curve segment starting at the first control point,  $\mathbf{P}_0$ , and chopping off the subsegments with deviation equal to flatness. The end points of these Bézier subsegments will define the point sequence of the approximating polyline.

Let a cubic Bézier curve segment  $\mathbf{Q}(t)$  be defined by its control points  $\mathbf{P}_{0..3}$ . The first step will be to store the first control point,  $\mathbf{P}_0$ , in the polyline point sequence list. Next we check the deviation of the curve against the required flatness,  $f$ . If it satisfies the flatness,  $d_{\perp}(\mathbf{Q}(\mathbf{p}_0, \mathbf{p}_3)) \leq f$ , then we can stop after storing the fourth control point,  $\mathbf{P}_3$ , in the polyline point list. If not, we then find the third control point's deviation from the  $r$ -axis of the normalized coordinate system.

Before we do that, we will have to translate the control point coordinates of the cubic Bézier curve segment to the new normalized coordinate system. Then we can plug in the deviation of the third control point in the formula,  $t = 2 \sqrt{\frac{f_r}{3P_{2_s}}}$  as derived in Section 4.3.

If the calculated  $t$  value turns out to be larger than 1, this would indicate the parabolic approximation is not suitable for this type of curve. In such relatively rare cases, recursive subdivision is used to generate the polyline approximation. The algorithm takes this path whenever the curve is drastically different from a parabola.

If the calculated  $t$  value is valid, i.e., in the interval (0,1), then we divide the curve at that value in two smaller Bézier subsegments. The first curve subsegment,  $S_{Q_1}$ , should have its deviation equal to required flatness,  $f$ . Since this calculation may not always be exactly right, we check its deviation against the required flatness and if we have a deviation a little larger than the flatness we use recursive subdivision to flatten the Bézier subsegment. Then we repeat the process on the second Bézier subsegment,  $S_{Q_2}$ . This process is continued until we have the second Bézier subsegment that satisfy the required flatness.



**Figure 17. Parabolic Approximation (PA) Algorithm Flow Chart**

Notice that the reduced flatness,  $f_r$ , is used for the calculation of the  $t$  value, but the deviation of different Bézier subsegments are checked to see if they satisfy the actual required flatness,  $f$ .

## **Chapter 5**

### **EMPIRICAL PERFORMANCE ANALYSIS**

In order to test our hypothesis we have implemented the devised algorithm and have compared its empirical performance against the competitive algorithm, recursive subdivision using the Hain stopping criteria [9]. The algorithm was implemented in C++, and the code listing is given in appendix C.

The parabolic approximating function is provided with inputs consisting of the control points of the Bézier curve segment, the required flatness, the percentage of the flatness used as the reduced flatness, and the array reference where the polyline coordinates are stored. The function returns the number of points added in the array, and the array is filled with the polyline coordinates. The recursive subdivision algorithm is also implemented and requires the same arguments with the exception of the reduced flatness.

#### **5.1 Testing Data Set**

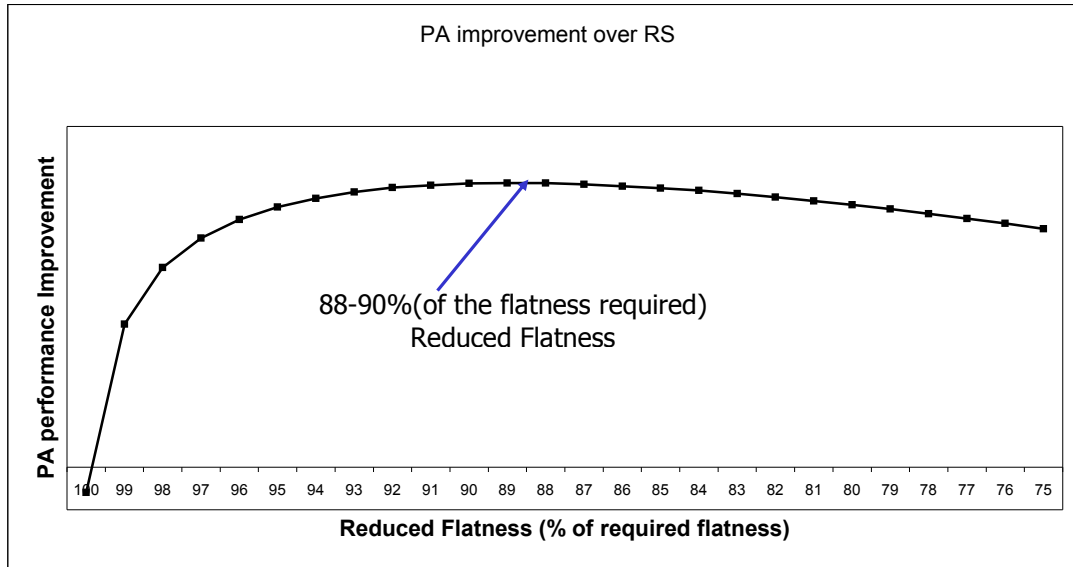
From test suites currently used for testing at MINOLTA-QMS, Inc., 6924 Bézier curves were extracted from different graphics pages. While most of these test suites consisted of Corel Draw and Acrobat Reader documents, we believe they represent a reasonably representative collection of graphic pages, and, in particular, curves. About 15.6% of the curves had inflection points.

## **5.2 Spatial Performance**

Spatial performance is measured by the number of line segments generated as an approximation of the Bézier curve. The smaller the number of line-segments the better the performance. According to hypothesis 1 (Section 3.1) we are aiming for a polyline approximation that has, on average, 15% fewer line-segments than that generated by the recursive subdivision (RS) with Hain's stopping criterion. Our results show that the amount of performance improvement is not only dependent on the flatness required but also on the reduced flatness.

### **5.2.1 Spatial Performance versus Reduced Flatness**

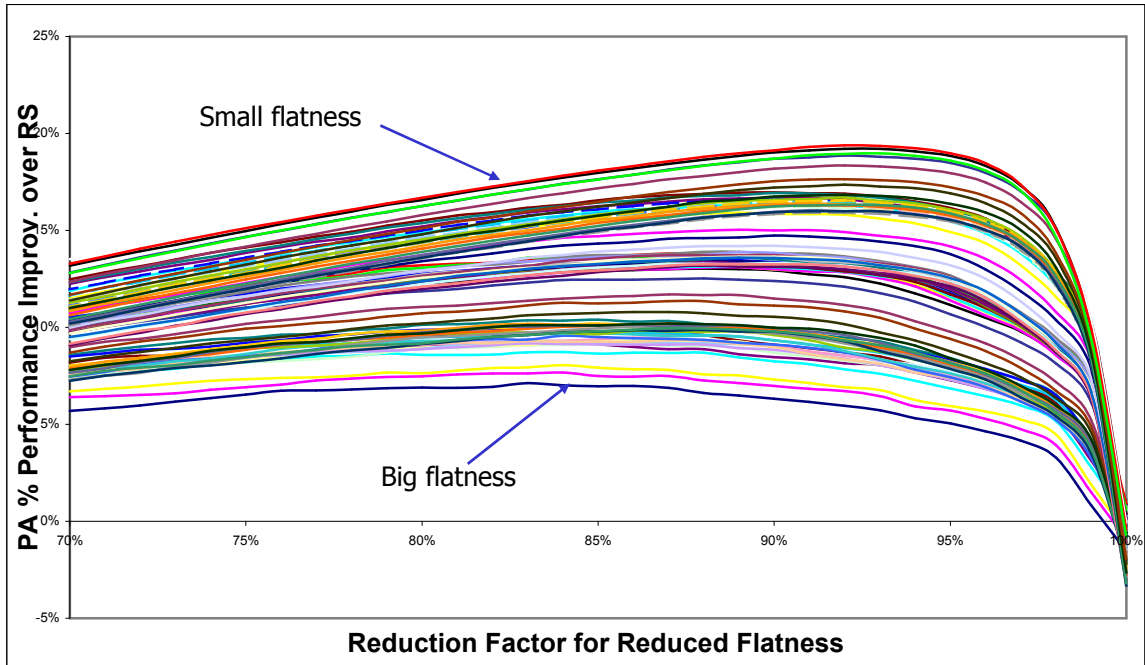
The graph in Figure 18 below shows the performance improvement in PA over RS with respect to the reduced flatness used by PA. Notice that in the graph the vertical axis is not labeled. This is because the performance is relative and the data used to plot this graph have different flatness. As can be seen, on average, the best performance is gained when the reduced performance is 88–90% of the required flatness.



**Figure 18. Average PA performance improvement against reduced flatness**

Recursive subdivision (RS) is better when the reduced flatness was not used by PA (i.e., 100% required flatness) because when the approximation generates a subsegment that has its deviation a little larger than flatness, we have to divide the subsegment again in flattening. The performance shoots up the moment we use even a small reduced flatness (99% of the required flatness.) It is reasonable to assume that the reason behind this is the arithmetical error involved in computation of the  $t$  value.

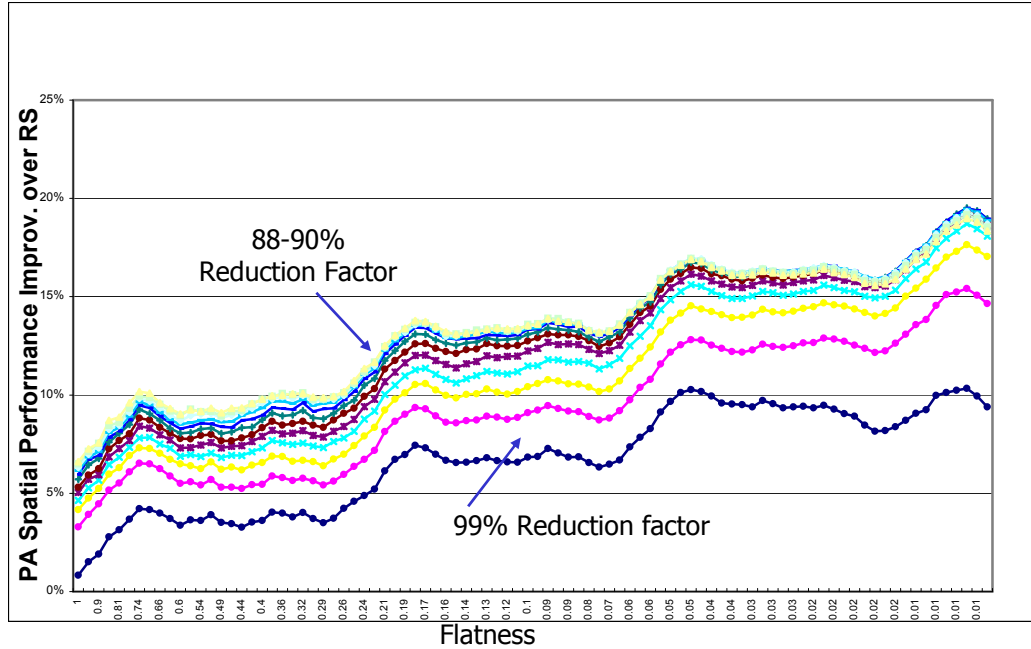
The chart in the figure below also show the change in the performance with respect to the reduced flatness, but each flatness is plotted as a different graph to quantify the PA performance gain.



**Figure 19. PA performance improvement against reduced flatness**

Note that for large flatness values the best performance gain is achieved with when 80–85% reduced flatness is used , while the best performance for smaller flatness is when the reduced flatness is 90–95%.

In Figure 20 below we have plotted the PA percentage performance improvement over RS against different flatness that we used for testing. Separate graphs for each reduced flatness are shown. Notice that the whole graph shifts up as we decrease the reduced flatness and the highest performance gain is when the reduced flatness is about 88–90%, in accordance with the overall value.



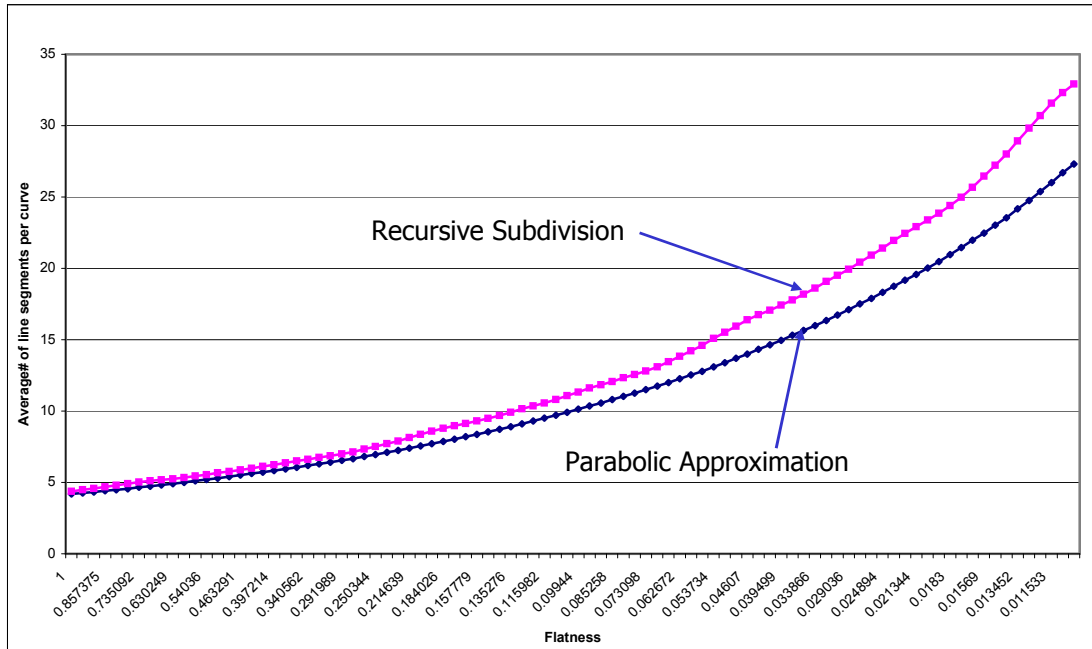
**Figure 20. Spatial Performance of different flatness**

### 5.2.2 Spatial Performance Vs Flatness

In Figure 20 above, notice that the performance is indirectly proportional to the flatness; as we reduce the flatness the performance increases. Flatness values ranging from 1.0 to 0.01 pixels are plotted on a logarithmic horizontal axis.

After a careful analysis, it was noted that the flatness that is sufficient for human eye, ranges from 0.4-0.6 pixel. The above graph tells us that PA is about 8–9% better than RS for these flatness. We claim in our hypothesis that the devised algorithm will have 15% better performance. While we did not get that much performance improvement for commonly used flatness (only 8–9%,) for flatness below 0.05 pixel the improvement is 15% or more. Moreover, it is reasonable to say that the performance improvement of PA over RS for our data set asymptotes to the estimated 25%.

Figure 21 shows the average number of line-segments generated by RS and PA (with reduced flatness 97%) for all different flatness tested. The gap between the curves widens as we decrease the flatness. Also notice that the parabolic approximation curve is smooth, while the recursive subdivision graph is not smooth. Since the performance improvement is the ratio between the two, these bumps are very prominent and seen as steps in Figure 20.



**Figure 21. Spatial Performance – RS v PS**

### 5.3 Temporal Performance

Timings data was calculated by flattening all ~7000 practical Bézier curves 50 times. Timings were performed on a Pentium III 500 MHz with 128MB of RAM running Windows 98. The number of iterations of the timed algorithm was chosen so that the smallest measured time was greater than 4 seconds. This ensured that the MS clock precision ( $\sim 1/17$  second) would not affect the results by more than 2%

Temporal performance is measured by the running time of each algorithm to flatten the whole curve. The smaller the time taken by the algorithm the better is the performance. Just as in spatial performance, the temporal performance is dependant on both the specified flatness and the reduced flatness.

### 5.3.1 Temporal Performance versus Reduced Flatness

According to hypothesis 2 in Section 3.1, we expected to be able to decrease the size of the approximating polyline, but only at the cost of calculation overhead. We set out goals to have this computational penalty no more than 100% of the time taken by the recursive subdivision with Hain [9] stopping criteria.

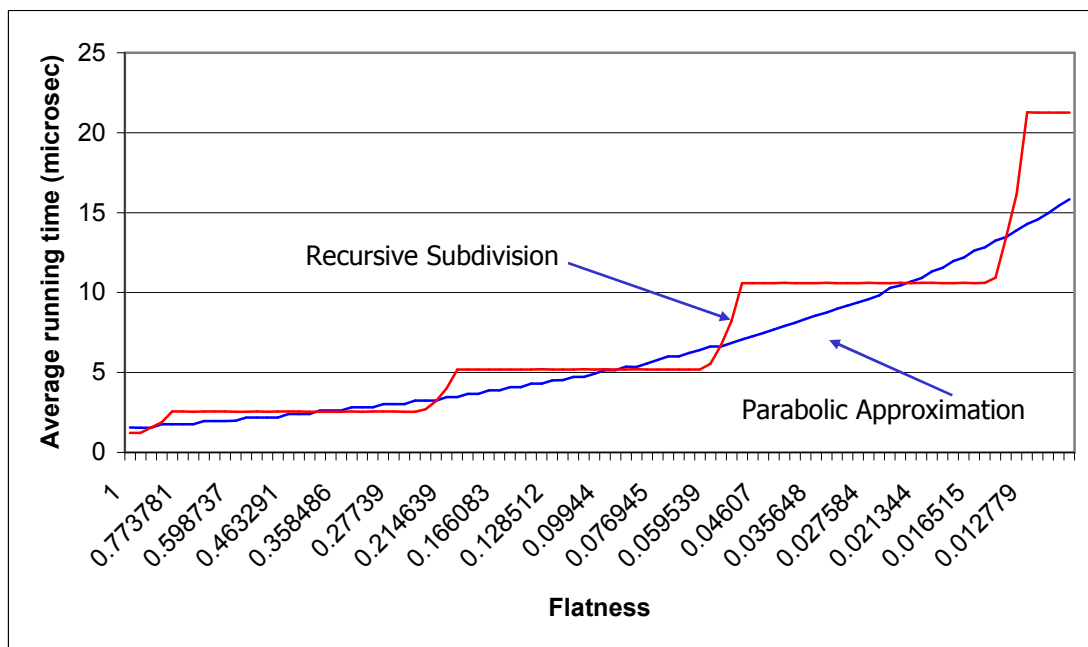


**Figure 22. Temporal Performance – PA improvement v reduced flatness**

However, according to the timing data collected is it clear that the devised algorithm is actually faster than RS with the Hain stopping criteria. Figure 23 shows the results. The average time taken by a Bézier curve using RS for flattening is 62 milliseconds (not shown in graph.) On the other hand, the time taken by the PA ranges from 56 to 60 milliseconds depending upon the reduced flatness used. The smallest time is taken by the PA when the reduced flatness is 97% of the required flatness. Therefore we have temporal performance improvement in PA as opposed to the expected calculation overhead.

### 5.3.2 Temporal Performance versus Flatness

It was also found that the the time taken to flatten the curve monotonically increases with its flatness. The smaller the flatness the larger the running time for the flattening algorithm This seems reasonable since the approximation with smaller flatness will have relatively more line-segments in it. Figure 23 shows the average running time versus flatness. The horizontal axis has flatness on a logarithmic scale. For simplicity, only 97% reduced flatness graph is shown for PA. The average time taken by the PA increases gradually as we decrease flatness over a logarithmic scale.



**Figure 23. Temporal Performance – PA and RS running times versus flatness**

On the other hand, the RS graph increases in a stepwise manner. Since RS is a recursive algorithm, and curves are subdivided in half until they satisfy flatness, it is reasonable to assert that the graph goes to the next step every time a new level of recursion is introduced. Also notice that, on average, the PA graph is lower than RS, meaning its rendering time is less, hence yielding an overall improvement over RS. This could be partly due to the fact that the parabolic approximation is an iterative rather than recursive, as in the case of RS.

## 5.4 Discussion

Our goal was to reduce the size of the display list by reducing the number of line-segments in the polyline approximation of a cubic Bézier curve segment. We were expecting to gain some spatial performance improvement at the cost of some amount of time which we hop to gain back by having fewer primitives to render in display list. However, we had a win-win situation, as not only we have reduced the size of the segment lis in our polyline approximation, but we have achieved a temporal performance improvement as well. The improvement is dependent on the flatness specification as well as the reduced flatness.

## **Chapter 6**

### **CONCLUSION**

#### **6.1 Summary of current work**

This research has developed an algorithm that generates a polyline approximation of the cubic Bézier curve segment. Since the algorithm approximates a cubic curve subsegment by a parabola, it is called parabolic approximation (PA). This algorithm generally reduces, and certainly never increases by more than 1, the number of line-segments generated by the widely used recursive subdivision (RS) algorithm (using the Hain stopping criterion, which is by itself already a significant improvement.) The reduction in the number of line-segments, i.e., the spatial performance of the algorithm, is dependent on the flatness specification. The spatial performance is indirectly proportional to the flatness and at a practical operating point (0.4–0.6 pixel flatness) it is about 8–9% better than the counterpart RS, while the improvement for smaller flatness (less than ~0.1 pixel) can be as much as 25%. Consequently, as higher resolutions are implemented in printers, there will be increasing benefit from this algorithm. The PA algorithm not only offers the spatial performance improvement, but also is faster than RS, i.e., offering temporal performance improvement. On average, the temporal performance gain is about 10-12%. So this is a win-win situation where we are not only reducing the size of the display list but also doing so in less running time.

## 6.2 Future work

The current research does not guarantee the least possible number of line segments. This is mainly because we are trying to interpolate the curve and limiting our selves to find the next point on the curve such that the corresponding subsegment's maximum transverse deviation is equal to flatness. Figure 4 shows that when same curve when extrapolated produce one less number of line segment in the approximation. However, solving extrapolation problem is not easy, as we introduce more degrees of freedom in that the calculation for the start and end points of approximating line-segments may not lie on the curve.

A second interesting research topic is the elimination of the flatness test after subdividing the curve for the calculated  $t$  value in parabolic approximation. This can be done if we can categorize the Bézier curve by its control points, and deciding *a priori* whether a parabolic approximation is in fact feasible.

## REFERENCES

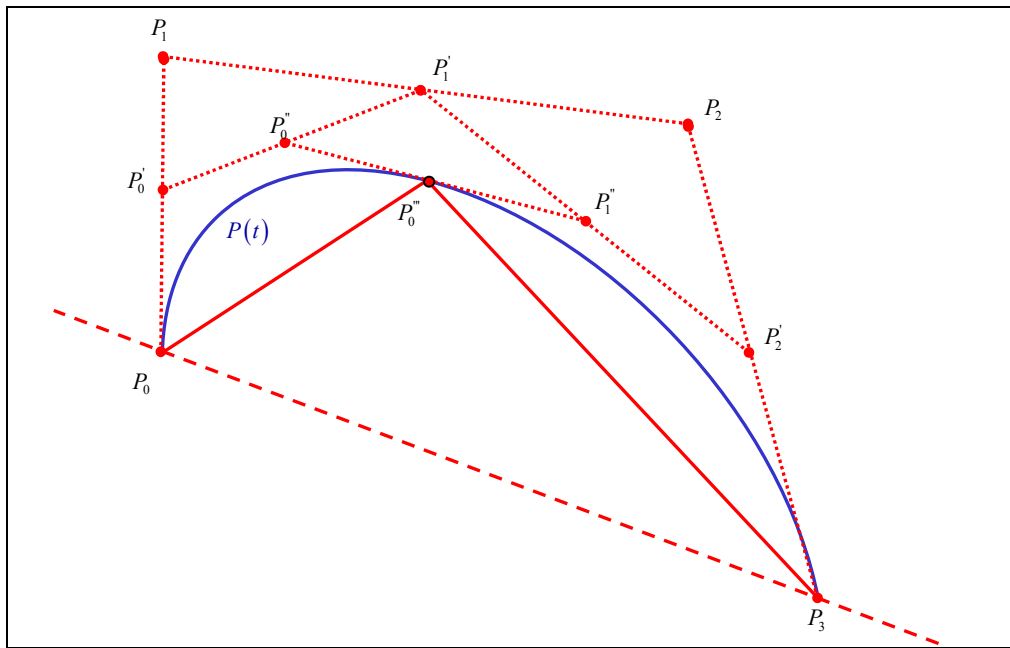
- [1] Adams, J. Alan, David F. Rogers, *Mathematical Elements For Computer Graphics*, McGraw-Hill Publishing Company, 1990, 291.
- [2] Barsky, B. and T. DeRose, "The Beta2-Spline: A Special Case for Beta Spline Curves and Surface Representation,": *CG & A*, 5(9), September 1985, Pages 56-58.
- [3] Barsky, B., T. DeRose, and M. Dippé, An Adaptive Subdivision Method with Crack Prevention for Rendering Beta-Spline Objects, Report UCB.CSD 87/348, Department of Computer Science, University of California, Berkeley, CA, 1987.
- [4] Bézier, P., *Emploi des Machines à Commande Numérique*, Masson et Cie, Paris, 1970. Translated by Forrest, A.R., and A. F. Pankhurts as *Bézier, P., Numerical Control – Mathematics and Applications*, Wiley, London, 1972.
- [5] Bézier, P., Mathematical and practical possibilities of UNISURF," in Barnhill, R. E., and R. F. Riesenfeld, eds., *Computer Aided Geometric Design*, Academic Press, New York, 1974.
- [6] De Casteljau, F., *Outillage Méthodes Calcul*, André Citroen Automobiles SA, Paris, 1959.
- [7] Foley, D.J., A. Van Dam, S.K. Feiner, and J.F. Hughes, *Computer Graphics Principles and practice*, Addison Wesley, 1995.
- [8] Gunther, Oliver, and Salvador Dominguez, "Hierarchical schemes for curve representation" *IEEE Computer Graphics and Applications*, May 1993 v13 n3 p55(9).
- [9] Hain. T.F., "Fast Termination Criterion for Recursive Subdivision of Bézier Curves", *37th ACM Southeast Conference*, Mobile, AL, April 15–18, 1999, pp 74–78.
- [10] Lane, J., and L. Riesenfeld, "A theoretical development for the computer generation of piecewise polynomial surfaces," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, PAMI-2(1), Jan 1980, 36–46.

- [11] Lien, S.L., M. Shantz, and V. Pratt, "Adaptive Forward Differencing for Rendering Curves and Surfaces," *Proceedings of SIGGRAPH '87* (Anaheim, California, July 27-32, 1987). In *Computer Graphics, Vol. 21, #4, July 1987*, ACM SIGGRAPH, New York. Pages 111-118.
- [12] Sederberg, Thomas W., Rida T. Farouki, "Approximation by Interval Bézier Curves" *IEEE Computer Graphics and Applications*, September 1992 v12 n5 p8(9).
- [13] Shantz, M., and S. Lien, "Rendering Trimmed NURBS with Adaptive Forward Differencing," *Proceedings of SIGGRAPH '89* (Boston, Massachusetts, July 31-August 4, 1989). In *Computer Graphics, Vol. 23, #3, July 1989*, ACM SIGGRAPH, New York. Pages 189-198.
- [14] Shantz, M., and S. Lien, "Shading Bicubic Patches," *Proceedings of SIGGRAPH '87* (Anaheim, California, July 27-32, 1987). In *Computer Graphics, Vol. 21, #4, July 1987*, ACM SIGGRAPH, New York. Pages 189-196.
- [15] Van Aken, Jerry, and Mark Novak, "Curve-Drawing Algorithms for Raster Displays" *ACM Transaction on Graphics*, v4, n2, p147-169, April 1985.
- [16] Wang, Guojin, and Wei Xu, "The Termination Criterion for Subdivision of the Rational Bézier Curves", *CVGIP: Graphical Models and Image Processing*, v53, n1, p93-96, January 1991.

## APPENDIX A

### Analytical Solution Work

Let  $\mathbf{P}_0, \mathbf{P}_1, \mathbf{P}_2$ , and  $\mathbf{P}_3$  be the control points of the Bézier curve segment  $\mathbf{P}(t)$ , as shown in the Figure 24, such that  $\mathbf{P}(t_0) = \mathbf{P}_0'''$  and  $d_{\perp}(S_p(\mathbf{P}_0, \mathbf{P}_0', \mathbf{P}_0'', \mathbf{P}_0''')) = f$ .

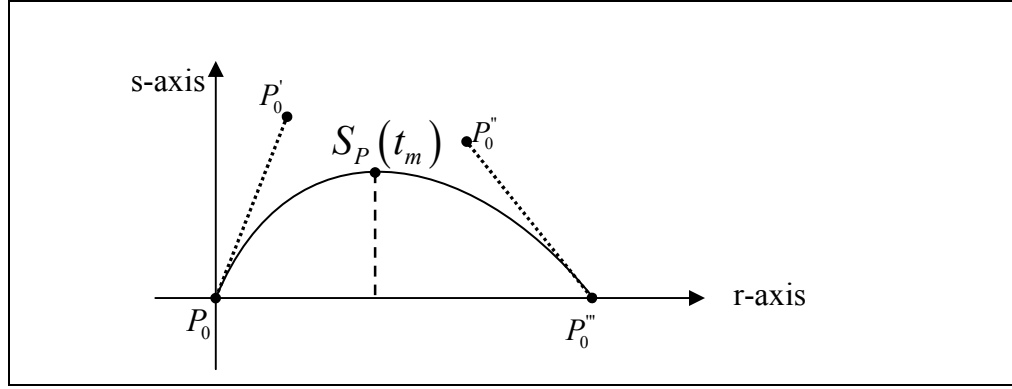


**Figure 24. Subdivision of Curve  $P(t)$**

In order to solve the problem we will have to find the value of  $t$  in terms of known variables such as control points of the Bézier curve. Four control points of the Bézier subsegments  $S_p$  can be found by the following equations[7].

$$\begin{aligned}
P_0' &= P_0 + t(P_1 - P_0) \\
P_1' &= P_1 + t(P_2 - P_1) \\
P_2' &= P_2 + t(P_3 - P_2) \\
P_0'' &= P_0' + t(P_1' - P_0') \\
P_1'' &= P_1' + t(P_2' - P_1') \\
P_0''' &= P_0'' + t(P_1'' - P_0'')
\end{aligned}$$

Consider a new coordinate system with its origin at  $\mathbf{P}_0$  and one of its axes (r-axis) is along the line-segment  $\overline{\mathbf{P}_0\mathbf{P}_0''}$ , while s-axis is orthogonal to it in a right hand rule. The new system is shown in the Figure 25.



**Figure 25. Bézier subsegment  $S_p$**

Maximum deviation of the subsegment occurs at  $t_m$ , where  $t_m \in [0,1]$ , such that

$S_{P_s}(t_m) = d_{\perp}(S_{P_s}) = f$ . Slope of the line tangential to the subsegment at  $t_m$ , in the new coordinate system where  $P_{0_s} = P_{0_s}'' = 0$  is given as:

$$\frac{d(S_{P_s}(t_m))}{dt_m} = (3 - 12t_m + 9t_m^2)P_{0_s}' + (6t_m - 9t_m^2)P_{0_s}'' = 0$$

Let  $v = P_{0_s}''/P_{0_s}'$ , which gives us the quadratic equation:

$$9t_m^2(1-v) - 6t_m(2-v) + 3 = 0$$

Solving for  $t_m$

$$t_m = \frac{(2-\nu) \pm \sqrt{\nu^2 - \nu + 1}}{3(1-\nu)}$$

Plugging value of  $\nu$  back in the equation gives us:

$$t_m = \frac{P_{0_s}'' - 2P_{0_s}' \pm \sqrt{(P_{0_s}'')^2 - P_{0_s}''P_{0_s}' + (P_{0_s}')^2}}{3(P_{0_s}'' - P_{0_s}')}$$

According to the Bézier subsegment equation:

$$S_{P_s}(t_m) = P_{0_s} + (3P_{0_s}' - 3P_{0_s}'')t_m + (3P_{0_s}'' - 6P_{0_s}' + 3P_{0_s})t_m^2 + (P_{0_s}'' - 3P_{0_s}' + 3P_{0_s} - P_{0_s})t_m^3$$

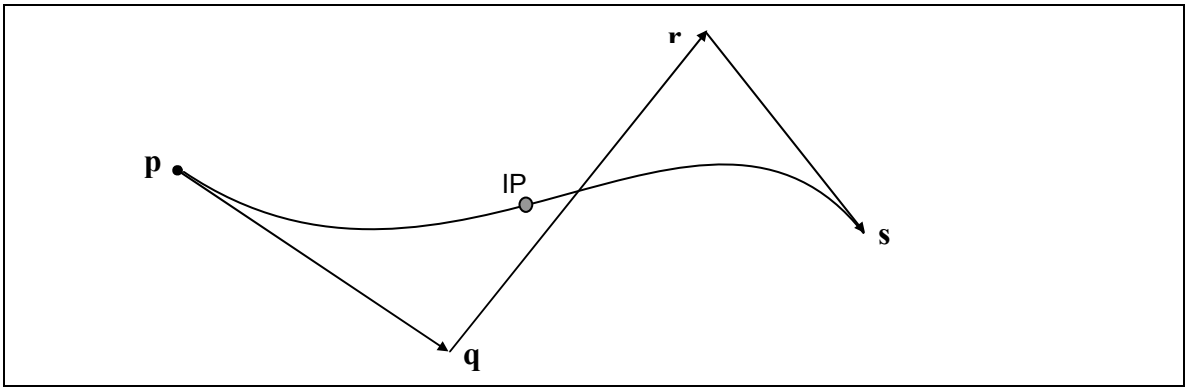
If we plug in the value of  $t_m$  and equate the above equation to the flatness  $f$ , after simplification we get:

$$\begin{aligned} & 6(P_{0_s}'')^4 - 6(P_{0_s}')^4 + (P_{0_s}'')^3(4P_{0_s}'' - 15P_{0_s}' + 14P_{0_s}) + (P_{0_s}')^3(15P_{0_s}'' - 14P_{0_s}' - 4P_{0_s}) \\ & + (P_{0_s}')^2 P_{0_s}''(21P_{0_s}'' + 15P_{0_s}') + 21(P_{0_s}'')^2 P_{0_s}'(P_{0_s}'' - P_{0_s}') \\ & + \sqrt{(P_{0_s}'')^2 - P_{0_s}''P_{0_s}' + (P_{0_s}')^2} \left[ 6(P_{0_s}')^3 - 6(P_{0_s}'')^3 + (4P_{0_s}'' - 12P_{0_s}' - 13P_{0_s}'')(P_{0_s}')^2 \right. \\ & \left. + (13P_{0_s}'' + 12P_{0_s}' - 4P_{0_s}'')(P_{0_s}'')^2 + 13P_{0_s}' P_{0_s}''(P_{0_s}'' - P_{0_s}') \right] \\ & = 27f(P_{0_s}'' - P_{0_s}')^3 \end{aligned}$$

Now we need to plug in the four control points of the Bézier subsegment in the equation above. We know these control points in terms of the control points of the actual Bézier segment and the unknown  $t$  value. Doing so will give us a polynomial with the unknown variable as  $t$ . The solution will be expensive to calculate. Moreover, since it is a polynomial of a high degree, we could have more than one unambiguous solution, which will further complicate matter. Therefore, the analytical solution will not be feasible for the algorithm.

## APPENDIX B

### Calculating Inflection Points



**Figure 26. Inflection point on Bézier curve**

Let  $\mathbf{p}$ ,  $\mathbf{q}$ ,  $\mathbf{r}$  and  $\mathbf{s}$  be the four successive control points of a Bézier curve. The number of inflection points can be evaluated by the following conditions:

```

if  $\left( \left( (\overline{\mathbf{pq}} \times \overline{\mathbf{qr}}) \cdot \hat{\mathbf{k}} \right) \left( (\overline{\mathbf{qr}} \times \overline{\mathbf{rs}}) \cdot \hat{\mathbf{k}} \right) < 0 \right)$ 
    // there is exactly one inflection point
else if  $\left( \left( (\overline{\mathbf{pq}} \cdot \overline{\mathbf{qr}}) > 0 \right) \wedge \left( (\overline{\mathbf{qr}} \cdot \overline{\mathbf{rs}}) > 0 \right) \right)$ 
    // the most common case with guaranteed no inflection point
else if  $\left( \left( (\overline{\mathbf{pq}} \times \overline{\mathbf{qr}}) \cdot \hat{\mathbf{k}} \right) \left( (\overline{\mathbf{pq}} \times \overline{\mathbf{rs}}) \cdot \hat{\mathbf{k}} \right) > 0 \right)$ 
    // some remaining cases with guaranteed no inflection point
else
    // (rare situation) there may be zero or two (possibly coincident) inflection points

```

Where  $\hat{\mathbf{k}}$  is a unit vector along the  $z$ -axis.

The first condition says that  $\mathbf{p}$  and  $\mathbf{s}$  are on opposite sides of the line through  $\overline{\mathbf{qr}}$  (be careful if  $\mathbf{q}$  and  $\mathbf{r}$  are coincident). Here a single inflection point is guaranteed since the curves curvature is first in one and then in the opposite direction. The condition can be calculated as follows:

$$\left( (q_x - p_x)(r_y - q_y) - (q_y - p_y)(r_x - q_x) \right) * \left( (r_x - q_x)(s_y - r_y) - (r_y - q_y)(s_x - r_x) \right) < 0$$

Here, the second condition filters out the predominant number of cases with no inflection point (those where the projection of the intersection of the lines through  $\overline{\mathbf{pq}}$  and  $\overline{\mathbf{rs}}$  onto the line through  $\overline{\mathbf{qr}}$  lies on the line segment  $\overline{\mathbf{qr}}$ , implying relative symmetry in the curve). This condition may be calculated by:

$$\left[ (q_x - p_x)(r_x - q_x) + (q_y - p_y)(r_y - q_y) > 0 \quad \wedge \quad (r_x - q_x)(s_x - r_x) + (r_y - q_y)(s_y - r_y) > 0 \right]$$

This is cheaper to evaluate than third condition, given below.

The third condition says that the intersection of the lines through  $\overline{\mathbf{pq}}$  and  $\overline{\mathbf{rs}}$  are on the opposite side of the line through  $\overline{\mathbf{qr}}$  from the points  $\mathbf{p}$  and  $\mathbf{s}$  (which are both on the same side of that line). This condition can be calculated similarly as above. This guarantees an arc (with no inflection points), and is calculated by:

$$\left( (q_x - p_x)(r_y - q_y) - (q_y - p_y)(r_x - q_x) \right) * \left( (q_x - p_x)(s_y - r_y) - (q_y - p_y)(s_x - r_x) \right) > 0$$

The remaining case may involve a curve with an arc or loop (no inflection points), a curve with two inflection points, or a cusp (two coincident inflection points.)

## APPENDIX C

### Code

```

/*****
 *   This code is the implementation of the Parabolic Approximation (PA). Although
 *   it uses recursive subdivision as a safe net for the failing cases, this is an
 *   iterative routine and reduces considerably the number of vertices (point)
 *   generation.
 *****/

#include <math.h>
#include "Bezier.h" // Including the Bezier class. Class has four vertices (control
                  // points i.e., b1, b2, b3 and b4) as its data members. Each
                  // vertex contains x and y coordinates. One of the member function
                  // of the bezier class worth mentioning finds the maximum
                  // deviation of the curve (maxDeviation).

static int noOfPoints; // Total number of points so far added to the list.
static float flatness; // required flatness
static float reducedFlatness; // reduced flatness

/*****
int parabolicApproximation( CBezier bez, // Bezier Control points
                          float f, // required flatness
                          CVertex polyLine[], // Array for storing the
                          // vertices (point)
                          float rFPer=1.0) // reduced Flatness
{
    // First we will set the static variables with the passed value for usage of other
    // functions. These variables are declared as static so better temporal performance.
    noOfPoints = 0;
    flatness = f;
    reducedFlatness = flatness * rFPer;

    int estimatedIFP = numberOfInflectionPoints(bez);
    if (estimatedIFP==0)
    {
        // If no inflection points then apply PA on the full Bezier segment.
        doParabolicApproximation(bez, polyLine);
        return noOfPoints;
    }

    // If one or more inflection point then we will have to subdivide the curve
    float t1, t2;
    int numOfIfP = findInflectionPoints(bez, t1, t2);

```

```

if (numOfIfP == 2)
{
    // Case when 2 inflection points then divide at the smallest one first
    CBezier sub1, tmp1, sub2, sub3;
    bez.subdivide(t1, sub1, tmp1);

    // Now find the second inflection point in the second curve an subdivide
    numOfIfP = findInflectionPoints(tmp1, t1, t2);
    if (numOfIfP == 2)
        tmp1.subdivide(t2, sub2, sub3);
    else if (numOfIfP == 1)
        tmp1.subdivide(t1, sub2, sub3);
    else
        return 0;

    // Use PA for first subsegment
    doParabolicApproximation(sub1, polyLine);

    // Use RS for the second (middle) subsegment
    useRecursiveSubdivision(sub2, polyLine);

    // Drop the last point in the array will be added by the PA in third subsegment
    noOfPoints--;

    // Use PA for the third curve
    doParabolicApproximation(sub3, polyLine);
}
else if (numOfIfP == 1)
{
    // Case where there is one inflection point, subdivide once and use PA on
    // both subsegments
    CBezier sub1, sub2;
    bez.subdivide(t1, sub1, sub2);
    doParabolicApproximation(sub , polyLine);
    noOfPoints--;
    doParabolicApproximation(sub2, polyLine);
}
else
    // Case where there is no inflection USA PA directly
    doParabolicApproximation(bez, polyLine);

return noOfPoints;
}

```

```

/*****/
// Recursive subdivision routine. This will not add the first point in the polyline
// approximation (first control point) in the list

void useRecursiveSubdivision(CBezier bez,          // Bezier curve
                             CVertex polyline[]) // Array containing the point
{
    double maxDev = bez.maxDeviation(); // Find the deviation of the curve

    if (maxDev > flatness)              // If not flattened yet then subdivide
    {
        CBezier b1, b2;
        bez.subdivide(0.5, b1, b2);
        useRecursiveSubdivision(b1, polyline);
        useRecursiveSubdivision(b2, polyline);
    }
    else // If (sub)curve is flattened then stop the point and stop
    {
        noOfPoints++;
        polyline[noOfPoints] = CVertex(bez.p4);
    }
}

/*****/
// Find the third control point deviation form the axis

float thirdControlPointDeviation(CBezier bez)
{
    float dx = bez.p2.x - bez.p1.x;
    float dy = bez.p2.y - bez.p1.y;
    float l2 = dx * dx + dy * dy;
    if (l2==0.0)
    {
        return 0.0;
    }
    float l = (float)sqrt(l2);

    float r = (bez.p2.y - bez.p1.y) / l;
    float s = (bez.p1.x - bez.p2.x) / l;
    float u = (bez.p2.x * bez.p1.y - bez.p1.x * bez.p2.y) / l;
    return ((float)fabs(r * bez.p3.x + s * bez.p3.y + u));
}

/*****/
// Find the number of inflection point

int numberOfInflectionPoints(CBezier bez)
{
    float dx21 = (bez.p2.x-bez.p1.x);
    float dy21 = (bez.p2.y-bez.p1.y);
    float dx32 = (bez.p3.x-bez.p2.x);
    float dy32 = (bez.p3.y-bez.p2.y);
    float dx43 = (bez.p4.x-bez.p3.x);
    float dy43 = (bez.p4.y-bez.p3.y);
    if ( ((dx21*dy32-dy21*dx32) * (dx32*dy43-dy32*dx43)) < 0)
        return 1; // One inflection point
    else if ( ((dx21*dx32+dy21*dy32)>0) ^ ((dx32*dx43+dy32*dy43)>0) )
        return 0; // Most cases no inflection point
    else if ( ((dx21*dy32-dy21*dx32) * (dx21*dy43-dy21*dx43)) > 0)
        return 0; // No inflection point
    else return -1; // cases where there in zero or two inflection points
}

```

```

/*****
// This is the main function where all the work is done

void doParabolicApproximation( CBezier bez,          // Bezier curve instance
                              CVertex polyLine[]    // Array to all the point approx
{
    // Add the first control point
    polyLine[noOfPoints] = CVertex(bez.p1);
    while (1)
    {
        double deviation = bez.maxDeviation();      // Find the deviation
        if (flatness > deviation)
        {
            // If the subsegment deviation satisfy the flatness then store the last
            // point and stop
            noOfPoints++;
            polyLine[noOfPoints] = CVertex(bez.p4);
            break;
        }

        // Find the third control point deviation and the t values for subdivision
        float d = thirdControlPointDeviation(bez);
        float t = 2 * (float)sqrt(reducedFlatness / d/3);

        if (t > 1)
        {
            // Case where the t value calculated is invalid so using RS
            useRecursiveSubdivision(bez, polyLine);
            break;
        }

        // Valid t value to subdivide at that calculated value
        CBezier b1, b2;
        bez.subdivide(t, b1, b2);

        // First subsegment should have its deviation equal to flatness
        deviation = b1.maxDeviation();
        if (deviation > flatness)
        {
            // if not then use RS to handle any mathematical errors
            useRecursiveSubdivision(b1, polyLine);
        }
        else
        {
            noOfPoints++;
            polyLine[noOfPoints] = CVertex(b1.p4);
        }

        // repeat the process for the left over subsegment.
        bez = b2;
    }
}

```

```

/*****/
// Find the actual inflection points and return the number of inflection points found
// if 2 inflection points found, the first one returned will be with smaller t value.

int findInflectionPoints( CBezier bez,      // Bezier curve
                        float &firstIfp,  // First inflection point reference
                        float &secondIfp) // Second inflection point reference
{
// For Cubic Bezier curve with equation  $P=a*t^3 + b*t^2 + c*t + d$ 
// slope of the curve  $dP/dt = 3*a*t^2 + 2*b*t + c$ 
//  $a = (\text{float})(-\text{bez.p1} + 3*\text{bez.p2} - 3*\text{bez.p3} + \text{bez.p4});$ 
//  $b = (\text{float})(3*\text{bez.p1} - 6*\text{bez.p2} + 3*\text{bez.p3});$ 
//  $c = (\text{float})(-3*\text{bez.p1} + 3*\text{bez.p2});$ 

float ax = (float)(-bez.p1.x + 3*bez.p2.x - 3*bez.p3.x + bez.p4.x);
float bx = (float)(3*bez.p1.x - 6*bez.p2.x + 3*bez.p3.x);
float cx = (float)(-3*bez.p1.x + 3*bez.p2.x);

float ay = (float)(-bez.p1.y + 3*bez.p2.y - 3*bez.p3.y + bez.p4.y);
float by = (float)(3*bez.p1.y - 6*bez.p2.y + 3*bez.p3.y);
float cy = (float)(-3*bez.p1.y + 3*bez.p2.y);

float a = (float)(3*(ay*bx-ax*by));
float b = (float)(3*(ay*cx-ax*cy));
float c = (float)(by*cx-bx*cy);

float r2 = (float)(b*b - 4*a*c);

firstIfp = 0.0;
secondIfp = 0.0;
if (r2>=0.0 && a!=0.0)
{
float r = (float)sqrt(r2);
firstIfp = (float)((-b + r) / (2*a));
secondIfp = (float)((-b - r) / (2*a));
if ((firstIfp>0.0 && firstIfp<1.0) && (secondIfp>0.0 && secondIfp<1.0))
{
if (firstIfp>secondIfp)
{
float tmp;
tmp = firstIfp;
firstIfp = secondIfp;
secondIfp = tmp;
}
if (secondIfp-firstIfp >0.00001)
return 2;
else return 1;
}
else if (firstIfp>0.0 && firstIfp<1.0)
return 1;
else if (secondIfp>0.0 && secondIfp<1.0)
{
firstIfp = secondIfp;
return 1;
}
}
return 0;
}
else return 0;
}
/*****/

```

## VITA

Athar Luqman Ahmad was born in Sargodha, Pakistan, on April 3<sup>rd</sup>, 1973. He graduated with Bachelors in Computer science from University of South Alabama in December 1997. He has 4½ years of professional experience including 1½ years of software quality assurance and 3 years of software development experience. He is currently working at MINOLTA-QMS Inc. as a software engineer. His responsibilities include the research and development of printer drivers and different print utilities for Microsoft Windows operating systems to support different manufactured laser printers. He has worked with COM objects, Windows API calls, classic Windows messaging and callback functions, and low level debugging tools such as Numega SoftIce and WinDbg. He has designed several user interfaces using MS Visual C++ for different projects and localized them in different languages including some far east languages. His hobbies include racket ball and hiking.