EG UK Theory and Practice of Computer Graphics (2011)
Hamish Carr, Ian Grimstead (Editors)

Pixel-Level Algorithms for Drawing Curves

Y.K. Liu[1], P.J. Wang[1], D.D. Zhao[1], D. Špelič[2], D. Mongus[2] and B. Žalik[2]

[1]College of Computer Science and Engineering, Dalian Nationalities University, Dalian 116600, China
[2]University of Maribor, Faculty of Electrical Engineering and Computer Science, Smetanova 17, SI-2000 Maribor, Slovenia

Abstract

This paper introduces new pixel level algorithms for parametric curves. The optimal value of the steps is first derived when rastrerizing parametric curves. A double-step algorithm using only integer arithmetic is presented to minimize the problem of oversampling. The algorithms for parametric curves have been implemented also on GPU. Through experiments we confirmed that the performance of this new algorithm is superior to previous algorithms.

Keywords: computing; curves; pixels; rasterization; GPU;

1. Introduction

Rasterization of geometric primitives was one of the first tasks for computer graphicians [Bre65]. Various approaches are known for rasterizing lines, circles and conics [AP92, Bre77, FH94, FH93, Mcl92, Pit85, Liu93a, Liu93b]. As they are placed at the end of the graphic visualization pipeline, they are frequently implemented in hardware by graphic output devices. Recently, research in curve rendering using graphics hardware has become attractive [LB05, RdMF08]. In this way, extreme speed is achieved and it seems it is purposeless thinking about improvements of the fundamental rasterisation algorithms. However, engineers still cope with the extremely limited environments in embedded systems. Especially high-end devices like domestic appliances, hi-fi sound systems, TV sets and cell phones are nowadays equipped with LCD displays, providing users with many additional features. Due the economic reasons, these devices does not have powerful graphical accelerators of today's desktop computers. Instead, LCD controllers provide only the most fundamental operations as turning the pixels on and off and refreshing the display. Therefore, the efficient and simply to implement rasterisation algorithms still plays important role in such environments.

This paper introduces a new algorithm for rasterizing curves. In Section 2, related work is presented. Section 3 presents a new algorithm for the rasterization of parametric curves, for which an optimum value of step-length is derived. The algorithm has been compared with known algorithms and have turned out to be considerably more efficient whilst the quality of the rasterisation remains the same. The algorithms for rasterizing the parametric curves have been implemented on Graphics Processing Unit (GPU), too. Results are presented in Section 4. Section 5 concludes the paper.

2. Existing pixel-level algorithms for rasterizing parametric curves

Pixel-level algorithms for drawing parametric curves have already been proposed [CSR89, Kla91, LSP87]. In the continuation, we briefly consider them and name them as traditional algorithms. These algorithms work in two steps:

- The length of the algorithm's step is determined during initialization. The algorithm generates $n+1$ points on the curve, therefore this step is calculated as $1/n$.
- During each iteration of the algorithm, an increment of coordinates on curve $C(t)$ is calculated and the current point is drawn. A difference method is used for the calculation. Assume that the k^{th} difference at the i^{th} point on the curve is expressed as $\Delta^k C(i)$, then from the difference formula

$$\Delta^{k+1}C(i) = \Delta^k C(i+1) - \Delta^k C(i) \tag{1}$$

we obtain

$$\Delta^k C(i+1) = \Delta^k C(i) + \Delta^{k+1} C(i); k = 0, \cdots, m. \tag{2}$$

DOI: 10.2312/LocalChapterEvents/TPCG/TPCG11/033-040

From the difference properties, we know that a polynomial of m^{th} order becomes constant after m difference operations. If all order differences at the i^{th} point are known, the $(i+1)^{th}$ point can be obtained by m additions. Therefore, by adding the first order difference at the i^{th} point to the function value at point $C(i)$, we can get the value of the function at the next point $C(i+1)$. The main loop of the algorithm is, therefore, as follows:

```
BEGIN
  FOR i = 1 TO n+1 DO
  BEGIN
    C_{i+1} = C_i + d_i;
    Setpixel();
    d_1 = d_1 + d_2;
    d_2 = d_2 + d_3;
        :
    d_{m-1} = d_{m-1} + d_m;
  END;
END.
```

where the variables can be declared as integers and, thus, only integer computations are needed.

Determining the magnitude of the algorithm step $1/n$'s is very important. The standard procedure for its determination is as follows: on the premise that if the magnitude of the curve's advance is less than the size of a pixel, the length of the step is selected to be as long as possible (i.e. the value of n is as small as possible), in order to speed up the algorithm. The so-called over-sampling problem occurs if the selected length of the step is too short. In this case, too many pixels have been drawn, the computation is lavish, and the rasterization time has slowed down. In traditional algorithms the value of n is estimated according to the curve's length:

$$n = max(nx, ny). \tag{3}$$

nx and ny are determined as

$$\begin{aligned} nx &= \sqrt{2}\, m \max_{0\le i\le m-1} |X_{i+1} - X_i|, \\ ny &= \sqrt{2}\, m \max_{0\le i\le m-1} |Y_{i+1} - Y_i|, \end{aligned} \tag{4}$$

where m is the order of the curve and (X_i, Y_i) are the coordinates of the curve's i^{th} control vertex. In the above formula, the multiplier $\sqrt{2}$ denotes the distance between two diagonally-adjacent pixels on the curve.

Huang and Zhu improved the above algorithm in two aspects [HZ00]: firstly, a more efficient integer method is proposed and, secondly, a better value for n is determined. This method is briefly summarised in the continuation. Suppose, the parametric curve equation is

$$x = f(t), \quad y = g(t), \quad 0 \le t \le 1. \tag{5}$$

Let us consider $x = f(t)$. The curve is divided into n segments (parameter t takes the values i/n, where $0 \le i \le n$). With the Intermediate Value Theorem, when n satisfies $n \ge \max_{0\le i\le 1} |f'(t)|$ we have

$$\left| f(\frac{i+1}{n}) - f(\frac{i}{n}) \right| = \frac{f'(\theta)}{n} \le 1. \tag{6}$$

It ensures that each step is less than a pixel and in this way guarantees the curve continuity. In order to use only integer calculations, the equation $x = f(t)$ is multiplied by a positive integer N, and becomes an integer equation

$$Nx_i = \phi(i) + z_i. \tag{7}$$

where $\phi(i) = Nf(\frac{i}{N})$ and its residue $z_i (|z_i| \le N/2)$ are integers. The meanings for each variable can be seen in Fig. 1. x_i denotes the pixel's x coordinate, the nearest to the calculated value of $f(\frac{i}{n})$. Residue z_i denotes the difference between x_i and $f(\frac{i}{n})$ (certainly, it has to be multiplied by N to become an integer). The case shown in Fig. 1a is characterized by the negative integer value of z_i, i.e., the pixel left of the actual curve is obtained. The opposite case, when z_i is a positive integer, is shown in Fig. 1b.

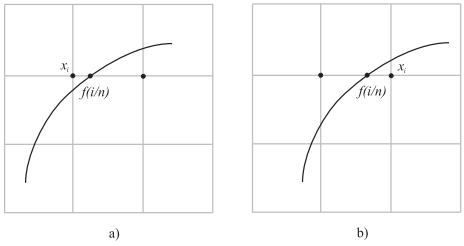

Figure 1: *Relationship between x_i and $f(i/n)$.*

Huang and Zhu's algorithm calculates points on the curve, pixel by pixel. Suppose the x coordinate of the current pixel on the curve is x_i. The x coordinate of the next pixel $x_i + 1$ should satisfy: $Nx_{i+1} = \phi(i+1) + z_{i+1}$, where $\phi(i+1) = \phi(i) + \Delta\phi(i) = Nx_i - z_i + \Delta\phi(i)$. From eq. (7) we have $|\Delta\phi(i)| = N|f(\frac{i+1}{n}) - f(\frac{i}{n})| \le N$, while $|z_i| \le \frac{N}{2}$, thus $|\Delta\phi(i) - z_i| \le \frac{3}{2}N$. The possible values for x_{i+1} are, therefore, $x_i - 1$, x_i or $x_i + 1$. Thus the formulae for x_{i+1} and z_{i+1} are obtained.

$$x_{i+1} = \begin{cases} x_i - 1 & , \ when \quad \Delta\phi(i) - z_i < -\frac{1}{2}N, \\ x_i & , \ when \quad -\frac{1}{2}N \le \Delta\phi(i) - z_i < -\frac{1}{2}N, \\ x_i + 1 & , \ when \quad \Delta\phi(i) - z_i \ge \frac{1}{2}N \end{cases} \tag{8}$$

$$z_{i+1} = \begin{cases} z_i - \Delta\phi(i) - N & , \quad when \quad x_{i+1} = x_i - 1, \\ z_i - \Delta\phi(i) & , \quad when \quad x_{i+1} = x_i, \\ z_i - \Delta\phi(i) + N & , \quad when \quad x_{i+1} = x_i + 1. \end{cases} \tag{9}$$

The formula for y coordinate is obtained similarly. The value of n obtained by this approach [HZ00] is smaller than that used by the traditional algorithm. It is still determined by eq. (3), but eq. (10) is used to determine nx and ny, instead of eq. (4).

$$\begin{aligned} nx &= m \cdot \max_{0 \le i \le m-1} |X_{i+1} - X_i|, \\ ny &= m \cdot \max_{0 \le i \le m-1} |Y_{i+1} - Y_i|, \end{aligned} \tag{10}$$

It is proved in [HZ00] that the value for n obtained in this way meets

$$n \ge \max_{0 \le t \le 1} |f'(t)|. \tag{11}$$

3. The algorithm for rasterizing parametric curves

3.1. The optimum value of n

When analyzing the results of the implemented Huang-Zhu algorithm, it was noticed that the maximal value of the step's length obtained by this algorithm varies within the range 0.6 to 0.8. This is, however, far from the optimum value 1. Therefore, our aim was to find the minimum value of n satisfying eq. (11), i.e. finding the minimum upper-bound of $|f'(t)|$. From the calculation process for n done in the previous subsection, and according to [HZ00], it is now known that $m \cdot \max_{0 \le t \le 1} |X_{i+1} - X_i|$ is the upper-bound for $|f'(t)|$, but it is not its minimum. In the continuation the minimum upper-bound is derived as follows.

The minimum upper bound for $|f'(t)|$ can appear at the extreme points or at the endpoints of the curve ($t = 0$ or $t = 1$). Therefore, the highest value of $|f'(t)|$ at one of the endpoints or extreme points is the minimum upper-bound. For determining n, we use a cubic Bézier curve as an example

$$f(t) = X_0(1-t)^3 + 3X_1 t(1-t)^2 + 3X_2 t^2(1-t) + X_3 t^3. \tag{12}$$

$$\begin{aligned} f'(t) = 3((X_3 - 3X_2 + 3X_1 - X_0)t^2 \\ +2(X_2 - 2X_1 + X_0)t + X_1 - X_0). \end{aligned} \tag{13}$$

By derivation, we obtain eq.(13). To get the extreme points for $f'(t)$, we set:

$$6((X_3 - 3(X_2 - X_1) - X_0)t + X_2 - 2X_1 + X_0) = 0. \tag{14}$$

Suppose that $V = X_2 - 2X_1 + X_0$, $W = X_3 - 3(X_2 - X_1) - X_0$. From the above equation, we know that $f'(t)$ reaches the extreme at $3(X_1 - X_0 - V^2/W)$ when $t = -V/W$. It is obvious that the value for $f'(t)$ at the ends ($t = 0$ and $t = 1$) are different $3|X_1 - X_0|$ and $3|X_3 - X_2|$. So the value of nx is defined by

$$nx = \begin{cases} 3\, max_a & , \quad when -\frac{V}{W} \in [0,1], \\ & , \\ 3\, max_b & , \quad when -\frac{V}{W} \notin [0,1], \end{cases} \tag{15}$$

$$max_a = max(|X_3 - X_2|, |X_1 - X_0 - \tfrac{V^2}{W}|, |X_1 - X_0|),$$

$$max_b = max(|X_3 - X_2|, |X_1 - X_0|).$$

Using the same approach, the value of ny is obtained from $ny \ge \max_{0 \le t \le 1} |g'(t)|$. Finally, we take

$$n = \max(nx, ny), \tag{16}$$

which gives the optimum value for n.

For conics, the extreme points of $f'(t)$ only appear at both ends. Therefore, the value for nx is $2\max(|X_2 - X_1|, |X_1 - X_0|)$. For cubics and quadrics, there should be one or two extreme points on the curve in addition to the ending points. Curves of higher order are used less. Table 1 shows the values of a number of iterations n obtained by different methods, while plotting two cubic Bézier curves, as shown in Fig. 2. Both methods determine the same set of pixels.

Table 1: *Comparison of the algorithms according to the value n*

	Traditional algorithm	Huang-Zhu algorithm	Improved algorithm
2a	849	600	480
2b	1188	840	600

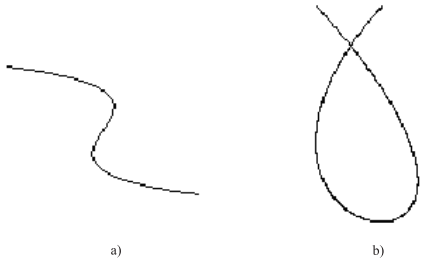

Figure 2: *Cubic Bézier curves.*

The curves shown in Fig. 2a and b consist of 170 and 294 pixels, respectively. The resolutions of the images have been reduced to clearly show the plotted pixels. Obviously, the number of iterations n for plotting one pixel is the smallest using our algorithm, which also affects the run-time. In the continuation, the so-called double-step pixel-level algorithm is proposed, which decreases the value of n further and achieves even higher speed.

3.2. A Double-Step Algorithm for Rasterizing Parametric Curves

In the previous subsection, the minimum value for n was derived by satisfying the condition of $n \geq \max_{0\leq t\leq 1}|f'(t)|$. If n is decreased further, discontinuous points could appear on the curve, as the length of the step may be longer than the pixel size. However, the basic idea of the proposed algorithm is to increase (doubling) the step-length (i.e., decreasing n by half). In this way, as the curve goes one step ahead, a pixel on the curve may be passed-over, and lost. In this case, the lost pixel is determined by interpolation. The benefit of this approach is obvious: the number of algorithm iterations is decreased by half, whilst the set of the plotted pixels is the same as for the single-step algorithm.

Assume n is the value of the steps for the single-step algorithm, as defined in the above paragraph. To make the algorithm double-step, eq. (6) should be multiplied by 2:

$$|f(\frac{i+1}{n/2}) - f(\frac{i}{n/2})| = |\frac{f'(\Theta)}{n/2}| \geq 2. \tag{17}$$

Eq. (17) ensures that the doubled step-length (i.e. the value of n decreases by half) is no longer than two-pixel lengths. In continuation, we consider how x coordinate is determined (y coordinate is obtained analogously). Suppose the current determined coordinate is x_i. With eq.(7), the next coordinate x_{i+1} should satisfy

$$Nx_{i+1} = \phi(i+1) + z_{i+1} \tag{18}$$

where

$$\phi(i+1) = \phi(i) + \Delta\phi(i) = Nx_i - z_i + \Delta\phi(i) \tag{19}$$

Since $\phi(i) = N\, f(\frac{i}{n})$, we know from eq. (17)

$$|\Delta\phi(i)| = N|f(\frac{i+1}{n/2}) - f(\frac{i}{n/2})| \leq 2N \tag{20}$$

while

$$|z_i| \leq \frac{N}{2} \tag{21}$$

thus

$$|\Delta\phi(i) - z_i| \leq \frac{5}{2}N. \tag{22}$$

It can be seen that the possible values for x_{i+1} are $x_i - 2$, $x_i - 1$, x_i, $x_i + 1$ and $x_i + 2$. According to eq. (8), the equation for calculating is obtained as follows:

When $(k - \frac{1}{2})N \leq \Delta\phi(i) - z_i < (k + \frac{1}{2})N$, then $x_{i+1} = x_i + k$, where $k = -2, -1, 0, 1, 2$. The value for residue z_{i+1} is determined as $z_{i+1} = Nx_{i+1} - \phi(i+1)$ (see eq. (9)). When $x_{i+1} = x_i + k$, then $z_{i+1} = z_i - \Delta\phi(i) + kN$, where $k = -2, -1, 0, 1, 2$. The recursive formula obtained in this way is then used to determine x coordinates of the next pixel and residue z for the next step.

When the increased or decreased amplitude of x coordinate is greater than the pixel length, i.e. ($x_{i+1} = x_i + 2$ or $x_{i+1} = x_i - 2$), the middle passed-over pixel is determined by the interpolation method. However, it is worth noticing that the occurrence probability for this is small (see the result in the next subsection). In this case, the following conditions have to be considered:

- if y coordinates do not differ ($y_{i+1} = y_i$), the middle pixel is trivially determined (x_{i+1}, y_i);
- if y coordinate is increased by 2 ($y_{i+1} = y_i + 2$), the middle pixel obviously has the coordinates $(x_i + 1, y_i + 1)$;
- if y coordinate increases by 1 ($y_{i+1} = y_i + 1$), then two pixels are considered, i.e. (x_{i+1}, y_i) and $(x_i + 1, y_i + 1)$. The nearer pixel to the curve is selected as the middle pixel. This pixel is determined by increasing the magnitude of y coordinate by half and testing whether its value is larger than $N/2$. If it is, then $(x_i + 1, y_i + 1)$ is selected, otherwise $(x_i + 1, y_i)$ is accepted.

4. Results

In this subsection, pixel-level curve-plotting algorithms are compared with other algorithms. As already shown (Table 1 in subsection 3.2), Huang-Zhu's algorithm is better than the traditional one, therefore only this algorithm was used for this comparison. Two versions of Huang-Zhu's algorithm were used: the basic one briefly described in subsection 3.1, and the improved one according to [HZ00]. This improved version does not plot repetitive pixels (i.e., after the location of the pixel is computed, the algorithm tests whether it is a repeated point; in this case, the pixel is not plotted). The algorithms were implemented in C. The results obtained when plotting curves from Fig. 2, are summarized in Table 2. As can be seen, the comparison was done according to three parameters:

- the number of algorithm cycles (n),
- the rate of the efficient point representing the redundancy degree of the points on the curve [Rap91] (some points are

calculated but they are not plotted, as they are rounded to the same pixels (these points are called inefficient points),
- spent CPU time plotting the curve 1000 times.

As seen from Table 1, the efficient point rate (EPR) of the double-step algorithm is much higher than for Huang-Zhu's algorithm. Because of this, the run-time of the proposed algorithm is shorter.

4.1. Implementation on GPU

Presented algorithms have also been implemented on a Graphics Processing Unit (GPU). GPUs have been developed to paralellize visualization of geometric entities on hardware. For this, various programming libraries and languages have been developed including CUDA [CUD11], GLSL [GLS11], and HLSL [HLS11]. HLSL stands for the High Level Shading Language for DirectX, and it has been applied in our case. Using HLSL, a C-like programmable shaders for the Direct3D pipeline can be created [HLS11]. Various versions of DirectX are available. In each version, new hardware capabilities are supported. The current version is 11 [GFW11]. DirectX version 10 has been applied, in our case due to the capability of the used graphics card (NVIDIA®GeForce®9800 GTX). DirectX 10 offers two shader types: vertex shader and pixel shader from previous versions, and it introduces a new geometry shader. The geometry shader can generate new graphics primitives such as points, lines, and triangles, which are from primitives sent into the graphics pipeline. GPUs are not very suitable for interative algorithms as proposed in this paper. In addition, GPUs unroll all loops and *if* statements [HB05], which would represents a bottle-neck in the total visualization time. As the algorithm for plotting the implicite functions consists of considerable number of if statemets in each iteration, its implementation on GPU is sensless. Because of this, we have only implemented parametric curves on GPU as follows.

The four control points of cubic Bézier curve are transferred to GPU through the vertex buffer pipeline. The points (pixels) on the curve are calculated in the geometry shader. Firstly, the number of pixels is determined as described in the paper. The geometry shader calculates the pixels and inserts them into the vertex buffer pipeline using Stream-Output Stage [MMS11]. Finally, the content of the vertex shader pipline is visualized. The procedure is schematically shown in Figure 3, while the HLSL code is given in the Appendix A. Table 2 shows the number of frames per seconds while plotting the curve. It is interesting to observe that Huang-Zhu's algorithm is faster that our algorithm although it plots less pixels. However, the double-step algorithm remains the fastest.

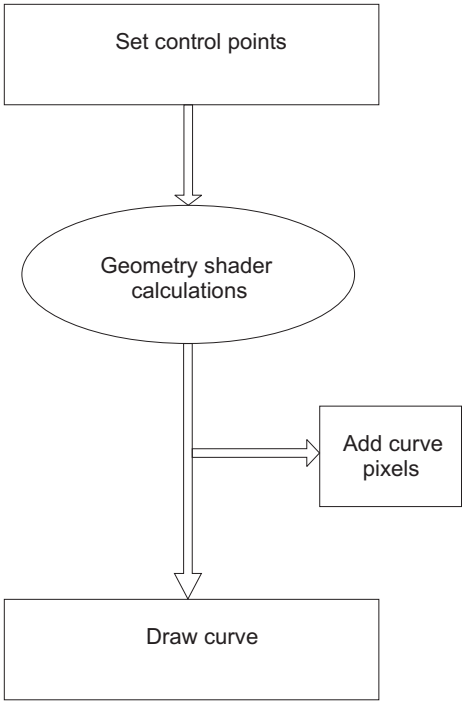

Figure 3: *Geometry shader work-flow.*

Table 3: *Frames per seconds achieved by GPU while plotting curves using different methods*

Control points	Huang-Zhu algorithm	Our algorithm	Double-step algorithm
C_1(-90,-90) C_2(100,100) C_3(-100,100) C_4(100,-100)	6200	6050	6890
C_1(-90,-90) C_2(-50,100) C_3(50,-100) C_4(100,100)	6180	6030	6720
C_1(-100,-100) C_2(100,-100) C_3(100,100) C_4(-100,100)	6170	6020	6720

5. Conclusion

This paper presents a new algorithm for rasterizing curves. The method deals with parametrically-defined curves. The crucial question, in this case, is how to determine the length of the step used by an algorithm. Oversampling is the most characteristic problem for these algorithms. In this paper, an optimum step-length is derived at, which minimise the oversampling problem. In this way, less cycles are needed by the algorithm, thus making it more efficient. However, a double-step algorithm using only integer arithmetic is introduced in order to further enhance the algorithm. The algorithms have been implemented also on GPU where the double-step algorithm remains the most efficient.

Table 2: *Comparison of the algorithms*

		Huang-Zhu algorithm		Huang-Zhu algorithm without repeated points		Double-step algorithm without repeated points	
		Figure 2a	Figure 2b	Figure 2a	Figure 2b	Figure 2a	Figure 2b
Value n	n	600	840	600	840	300	420
given by [HZ00]	EPR	28.3%	34.9%	28.3%	34.9%	56.6%	69.8%
	t(s)	2.36	3.35	0.79	1.32	0.74	1.26
The new	n	480	600	480	600	240	300
value of n	EPR	35.3%	48.9%	35.3%	48.9%	70.5%	97.7%
	t(s)	1.98	2.56	0.77	1.29	0.69	1.21

5.1. Acknowledgements

This work was supported by the National Natural Science Foundation of China (60675008), bilateral China-Slovene project *Study of Selected Algorithms on Computational Geometry and Shape Representation* (BI-CN/07-09/012) and Slovenian Research Agency under the grant 1000-08-31010.

References

[AP92] ANANTAKRISHNAN N., PIEGL L.: Integer de casteljau algorithm for rastering nurbs curves. *Computer Graphics Forum 11 (2)* (1992), 151–162. 1

[Bre65] BRESENHAM J. E.: Algorithms of computer control of a digital plotter. *IBM System Journal 4 (1)* (1965), 25–33. 1

[Bre77] BRESENHAM J. E.: A linear algorithm for incremental digital display of circular arcs. *Communications of the ACM 20 (2)* (1977), 100–106. 1

[CSR89] CHANG S. L., SHANTZ M., ROCCHETTI R.: Rendering cubic curves and surfaces with integer adaptive forward differencing. *Computer & Graphics 23(3)* (1989), 157–166. 1

[CUD11] CUDA: http://www.nvidia.com/object/what_is_cuda_new.html, March 2011. 5

[FH93] FELLNER D. W., HELMBERG C.: Robust rendering of general ellipses and elliptical arcs. *ACM Transactions on Graphics 12(3)* (1993), 251–276. 1

[FH94] FELLNER D. W., HELMBERG C.: Best approximate general ellipses on integer grids. *Computers & Graphics 18(2)* (1994), 143–151. 1

[GFW11] GFW: http://www.gamesforwindows.com/en-us/directx/, March 2011. 5

[GLS11] GLSL: http://www.opengl.org/documentation/glsl/, March 2011. 5

[HB05] HARRIS M., BUCK I.: Gpu flow-control idioms. In *GPUGems 2 – Programming Technique for High-performance Graphics and General-propose Computation* (2005), Pharr M., R.Fernando, (Eds.). 5

[HLS11] HLSL: http://msdn.microsoft.com/en-us/library/bb509561(v=vs.85).aspx, March 2011. 5

[HZ00] HUANG Y.-D., ZHU G.-Q.: A fast point-by-point generating algorithm for polynomial parametric curve. *Chinese Journal of Computers 23(4)* (2000), 393–397. 2, 3, 4, 6

[Kla91] KLASSEN R. V.: Integer forward differencing of cubic polynomials: analysis and algorithms. *ACM Transaction on Graphics 10(2)* (1991), 152–181. 1

[LB05] LOOP C., BLINN J.: Geometry on gpus: Resolution independent curve rendering using programmable graphics hardware. *ACM Transactions on Graphics 24(3)* (2005), 1000–1009. 1

[Liu93a] LIU Y.-K.: Algorithm for circle approximation and generation. *Computer-Aided Design 25(3)* (1993), 169–171. 1

[Liu93b] LIU Y.-K.: The generation of circular arcs on hexagonal grids. *Computer Graphics Forum 12(1)* (1993), 21–26. 1

[LSP87] LIEN S. L., SHANTZ M., PRATT V.: Adaptive forward differencing for rendering curves and surfaces. *Computer & Graphics 21(4)* (1987), 111–118. 1

[Mcl92] MCLLROY M. D.: Getting raster ellipses right. *ACM Transactions on Graphics 11(3)* (1992), 259–275. 1

[MMS11] MMS: http://msdn.microsoft.com/en-us/library/bb205121(v=vs.85).aspx, March 2011. 5

[Pit85] PITTEWAY M. L. V.: Algorithms of conic generation, fundamental algorithms for computer graphics. *NATO ASI Series F-17* (1985), 219–237. 1

[Rap91] RAPPOPORT A.: Rendering curves and surfaces with hybrid subdivision and forward differencing. *ACM Transaction on Graphics 10(4)* (1991), 323–341. 4

[RdMF08] RUEDA A. J., DE MIRAS J. R., FEITO F. R.: Gpu-based rendering of curved polygons using simplicial coverings. *Computers & Graphics 32* (2008), 581–588. 1

Appendix A: HLSL Source Code

```
// structure for Vertex shader data input
struct VS_IN { float4 pos : POSITION; };

// structure for Pixel shader data input
struct PS_IN { float4 pos : SV_POSITION; };

// structure for Geometry shader data input
struct GS_IN { float4 pos : POSITION; };

// Vertex data are obtained at the input
// the position of vertex data is returned
GS_IN VS(VS_IN input) {
    GS_IN output = (GS_IN)0;
    output.pos = input.pos;
    return output;
}

// Code for Pixel shader; each pixel is drawn in white
float4 PS(PS_IN input):SV_Target {return float4(1,1,1,1);}

// code for Geometry shader
void GSScene(line GS_IN input[2],
inout LineStream<PS_IN> OutputStream) {
// define output vertex variable
  PS_IN output = (PS_IN)0;
  float pX[4];
  float pY[4];
  int N = 0;
  int i = 0;

  // store control points in the array
  pX[0] = input[0].pos[0];
  pX[1] = input[0].pos[2];
  pX[2] = input[1].pos[0];
  pX[3] = input[1].pos[2];
  pY[0] = input[0].pos[1];
  pY[1] = input[0].pos[3];
  pY[2] = input[1].pos[1];
  pY[3] = input[1].pos[3];

  // calculate number of pixel
  float maxxn=0;
  float maxyn=0;
  [unroll] for(i=1; i<=3; i++) {
      if(pX[i]-pX[i-1] > maxxn) maxxn=pX[i]-pX[i-1];
      if(pY[i]-pY[i-1] > maxyn) maxyn=pY[i]-pY[i-1];
  }
  N = (max(maxxn,maxyn) * 3) / 2;
  float step=1.0/N;
  float t=0;

  // Calculation of Bezier curves
  float X_c[4];
  float Y_c[4];

  X_c[3]=pX[3];
  X_c[3]+=-3*pX[2];
  X_c[3]+=3*pX[1];
  X_c[3]+=-1*pX[0];

  X_c[2]=3*pX[2];
  X_c[2]+=-6*pX[1];
  X_c[2]+=3*pX[0];

  X_c[1]=3*pX[1];
  X_c[1]+=-3*pX[0];

  X_c[0]=pX[0];

  Y_c[3]=pY[3];
  Y_c[3]+=-3*pY[2];
  Y_c[3]+=3*pY[1];
  Y_c[3]+=-1*pY[0];

  Y_c[2]=3*pY[2];
  Y_c[2]+=-6*pY[1];
  Y_c[2]+=3*pY[0];

  Y_c[1]=3*pY[1];
  Y_c[1]+=-3*pY[0];

  Y_c[0]=pY[0];

  //Partial differencing
  float dx[4];
  float dy[4];
  float step2 = step * step;
  float step3 = step2 * step;

  dx[0]=X_c[0];
  dx[1]=X_c[1]*step+X_c[2]*step2+X_c[3]*step3;
  dx[2]=2*X_c[2]*step2+6*X_c[3]*step3;
  dx[3]=6*X_c[3]*step3;

  dy[0]=Y_c[0];
  dy[1]=Y_c[1]*step+Y_c[2]*step2+Y_c[3]*step3;
  dy[2]=2*Y_c[2]*step2+6*Y_c[3]*step3;
  dy[3]=6*Y_c[3]*step3;

  float X[1000];
  float Y[1000];
  X[0]=dx[0];
  Y[0]=dy[0];

  // send first pixel to be drawn
  output.pos = float4( X[0], Y[0], 0, 1);
  OutputStream.Append(output);

  [unroll] for(i=1; i < N; i++) {
      // for each pixel calculate x and y
      X[i]=X[i-1]+dx[1];
```

```
        dx[1]+=dx[2];
        dx[2]+=dx[3];

        Y[i]=Y[i-1]+dy[1];
        dy[1]+=dy[2];
        dy[2]+=dy[3];

        // send calculated position of pixel to OutputStream
        output.pos = float4( X[i], Y[i], 0, 1);
        OutputStream.Append(output);
        t+=step;
    } // Send GPU comand to draw all pixel in OutputStream
    OutputStream.RestartStrip();
}

// define Render technique
technique10 Render {
    pass P0 {
        SetGeometryShader(CompileShader(gs_4_0, GSScene() ) );
        SetVertexShader(CompileShader(vs_4_0, VS() ) );
        SetPixelShader(CompileShader(ps_4_0, PS() ) );
    }
}
```