Resolution Independent NURBS Curves Rendering using Programmable Graphics Pipeline

Rami Santina
CCT International
rsantina@cctintl.com

Abstract

Non-Uniform Rational B-Splines (NURBS) are widely used, especially in the design and manufacturing industry, for their precision and ability to represent complex shapes. These properties come at the cost of being computationally expensive for rendering. Many methods have tackled NURBS rendering by view based approximations and/or heavy preprocessing. We present a method for resolution independent rendering of curves and shapes, defined by NURBS, by utilizing the high parallelism of the programmable graphics hardware. The computation of the curve is processed directly on the GPU, without the need for complex preprocessing and/or additional storage of the basis functions as textures. Our method enables rendering of a complex NURBS shape in precise form, by defining only the curve's hull. We also present a method to enhance the performance of the preprocessing stage, mainly triangulation, that fits our requirements and speeds up the process. With optimized preprocessing and using only the mobile profile of the programmable graphics pipeline, we achieve a fast and resolution independent method for rendering NURBS based 2D shapes on desktop and mobile devices.

Keywords: *NURBS, Curve Rendering, Resolution Independence, GPU Algorithm, Mobile Graphics.*

1. INTRODUCTION

Resolution independent rendering is becoming a standard requirement for visualizing shapes. Earlier methods presented a resolution independent rendering for Bezier curves. The Bezier form, by definition, limits the curve's shape to the position of the control vertices. Thus, editing the control vertices is the only degree of freedom (DoF) for defining the curve. Such form of editing requires a regeneration of the shape and/or additional control vertices. In addition, designing some shapes using Bezier requires more control vertices and sometimes a higher order definition; this can be reduced by using a more general family of curves. Non-Uniform Rational B-Splines (NURBS) are widely used for a precise design of complex shapes using fewer control vertices, especially in CAD/CAM based applications. The evaluation of a NURBS curve is quite expensive requiring a recursive computation of the basis functions. Many tools transform, as a preprocessing step, the curves to a more simplified form to speed up the rendering.

In this paper, we present a method for Resolution Independent rendering of 2D shapes, defined partially or totally by NURBS curves, using the graphics hardware. Our method defines a simple and memory efficient approach to render 2D NURBS shapes by utilizing the high parallelism of the programmable graphics hardware. The NURBS curve is evaluated, using an implicit function, during rasterization.

The rest of this paper is organized as follows: Related Works are discussed in Section 2. . In Section 3., we present a general overview of our method describing the preprocessing stage and the GPU based NURBS curves rendering algorithm. In Section 4., we enhance the rendering technique to anti-alias the resulting curve. In Section 5., we discuss anti-aliasing for the whole shape to ensure generating a smooth and continuous representation.

2. RELATED WORKS

In [1], the authors presented a method for rendering NURBS curves on the GPU using textures, generated in a preprocessing phase, to store the values of the basis functions depending on the curve's order. This technique produces good results, but requires a computationally expensive preprocessing step and additional memory to store the textures.

To represent our NURBS shapes on the graphics hardware, we depend on the curve's implicit form [2]. In [3] and [4], implicit curve rendering has been used for representing curves and surfaces based on distance approximations. In [5], the authors presented a method for embedding sharp linear features into images to obtain resolution independence while leveraging GPU pixel processing. In [6], curved elements were embedded into texture images at the texel level. In these methods a computationally expensive preprocessing, performed on the CPU, was required.

In [7], the authors presented a method for rendering 2D regions based on quadratic Bezier curves. They defined each curve by a triangle, formed by the curve's control hull. Each of these triangles is rasterized using an implicit equation which defines the Bezier Curve. The mosaic formed by the set of such triangles along with the non-curved triangles gives the form of the final 2D shape. They then extended this process to handle cubic Bezier curves defined by neighboring triangles, depending on a classification of the curve. For preprocessing, the authors rely on constrained delaunay triangulation for generating the non-curved triangles; which is computationally expensive, in its general form. For anti-aliasing of the whole shape, they primarily depend on hardware accelerated Multi-Sampling. This technique produces good results but does not provide high quality anti-aliasing for tiny and skinny shapes, like small text, since the shape as a whole will be blurred.

One of the main applications for [7], is resolution independent font rendering. Earlier methods for font rendering are based mainly on generating a texture processed on the CPU [8], which produces crisp and sharp small text, but are resolution dependent. Our algorithm supports font rendering since the bezier curve is a special case of NURBS. In section 5. we will present a technique which produces high quality text, as an optional extension to our method.

3. ALGORITHM OVERVIEW

The input to our algorithm is a set of outlines which represents the shape's boundaries. Each of the outlines consists of a set of connected vertices. The vertices are of two types, namely: off-curve and on-curve (interpolated by the curve). One or more off-curve vertex defines a curve with endpoints being the closest neighboring on-curve vertices from the same outline. Examples of similar outline based definition are the 2D CAD drawings, scalar vector data, and fonts data (figure 1). In addition, each off-curve vertex has a weight, which defines the influence of the vertex on the curve's final shape. An off-curve vertex with no predefined weight is assigned a weight $w = 1$.

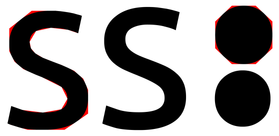


Figure 1: Snapshots of Rendered text, using the NURBS rendering described along this paper, with and without the discarded pixels of the curved triangles showing method compatibility with TTF rendering. **Left:** the glyph of character S from a TTF based font data. **Right:** A conic shape

We convert all the curved regions of an outline to a set of triplets, defined by two on-curve vertices surrounding an off-curve vertex. This representation enables us to map the curved regions to a set of *curved* triangles. For the **quadratic case**, the conversion is simply performed by representing each curve segment by the 3 vertices that influence the curve shape. As for the **cubic case**, we subdivide the curve to the quadratic form [9]. Generalizing the equations to cubic form is doable by formulating the equations in the same steps described along this paper, but will require a preprocessed classification of the curve's shape and a more computationally expensive rendering. The transformation to quadratic is a mathematical approximation. Since the rendered curve is accurate only to the pixel level, this approximation doesn't effect any details of the shape

With all the curved regions defined as a set of triangles, we map the rendering of the curves to texture space; assigning the curve control points as texture coordinates to the curve vertices, this step is detailed in Section 3.2. As for the non curved parts we perform a modified Delaunay triangulation (Section 3.1). Finally, the set of triangles generated by the triangulation along with the curved triangles form the final shape of the 2D object. An example output is shown in figure 2.

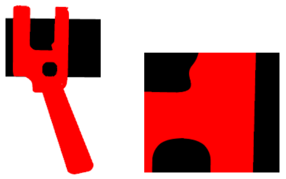


Figure 2: A CAD handle rendered using our method, with different weights. **Right:** Zoom in on part of the shape.

The produced shapes will have the following properties:

1. viewpoint independence, since the curves are computed on the GPU, using an implicit function, during the rasterization phase.

2. minimized memory usage, since we are not using subdivision to approximate the curved parts and we are not using any textures in the algorithm.

3. high performance rendering, since preprocessing (mainly triangulation) is computed once at input definition and rendering is processed directly on the GPU.

4. mobile compatible, since our method only uses features included in the mobile profile (OpenGL ES2)

3.1 Preprocessing

In this section, we present a modified version of constrained delaunay triangulation [10] that fits our requirements, removing the need for any cleanup stages. Note that, this is not a requirement for the algorithm described in this paper; but is a major block in the preprocessing phase which affects the overall performance and memory usage. As stated earlier, the curved regions are transformed into curved triangles which excludes them from the general triangulation step and thus from the algorithm described in what follows.

First, we transform each of the input outlines to a set of connected half edges forming a loop λ. Then, we add each λ, sequentially, to the final set of loops Λ. While adding, we check if this loop is intersecting, constrain, or is constrained by any of the already added loops in Λ, using a simple ray tracing (in/out) test. If the loop constrain or is constrained, we combine the two loops at the closest pair of vertices by adding two sibling half-edges between them. Hence, we get a simplification of the problem since now we have only one well connected loop. If the loop intersects, we split at the intersecting positions, which may result in a maximum of two new vertices. Then, we constrain the first by the contained part of the second and combine the two open ended parts forming another closed loop.

At the completion of the above steps, Λ contains a set of independent closed loops that define the resulting shape. Hence, we can now triangulate each loop separately without any additional cleanup steps.

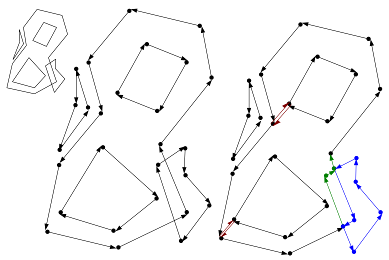


Figure 3: **Top Left:** Input shape defined as a set of connected vertices forming five outlines. **Middle:** Outlines converted to half edges. **Right:** The constraint outlines are merged with the respective outline (red half-edges), and the intersecting outlines are split editing the second large outline (green half-edges) and forming the third outline (blue half-edges)

In figure 3, the five input outlines are transformed into 3 well connected independent outlines. This step simplifies the triangulation process of the shape, to be a triangulation of independent set of loops, where a cleaning phase is no more required to define the holes. Note that we can not guarantee that a complete delaunay triangulation can be achieved, but we can maximize it using a greedy approach.

3.2 Quadratic NURBS Rendering

Since all the curved parts of the region have been transformed into quadratic NURBS, the following is applied to all the curved triangles.

The general form of a NURBS curve is given by,

$$C(x) = \frac{\sum_{i=0}^{n} N_{i,D}(x) w_i P_i}{\sum_{j=0}^{n} N_{j,D}(x) w_j} \tag{1}$$

where

$$N_{i,1}(x) = \begin{cases} 1 & \text{if } t_i \le x < t_{i+1} \\ 0 & \text{otherwise.} \end{cases}$$

and

$$N_{i,D}(x) = \frac{(x - t_i) N_{i,d-1}(x)}{t_{i+d-1} - t_i} + \frac{(t_{i+d} - x) N_{i+1,d-1}(x)}{t_{i+d} - t_{i+1}}$$

where t_i corresponds to the knot at location i in the Knot Vector.

Equation 1, gives the definition of a NURBS curve C as a function of the parameter x, where P_i are the control points and $N_{i,D}(u)$ are the basis functions of degree D. w_i are the weights of each control point. The special case of $\frac{0}{0}$ that may arise in one of the basis functions, is taken to be 0.

We first map the curve definition to texture space; by assigning the control points of our quadratic NURBS curve as attributes to the vertices. Hence, the control points p_0, p_1, and p_2 are assigned to the vertices by the set [u v w], where [u v] are the texture coordinates and w is the weight. We set $p_0 = [0\ 0\ w_0]$, $p_1 = [\frac{1}{2}\ \frac{1}{2}\ w_1]$ and $p_2 = [1\ 0\ w_2]$, where p_1 is assigned to the off-curve vertex. During rasterization, the GPU will calculate a texture coordinate for each pixel on the interior of the triangle by interpolating the defined texture coordinates.

Since u belongs to [0 1] and the curve is defined in the domain [0 1], we get the following property: for each value of u generated by the interpolation there exists a value v which is on the curve. In the fragment shader, we determine the fragment position w.r.t. the curve by evaluating the implicit function of the curve [2], which can be derived as:

$$f = v - \frac{w_1 u(1-u)}{(w_0 - 2w_1 + w_2)u^2 + 2(w_1 - w_0)u + w_0} \tag{2}$$

If $f < 0$, then the fragment belongs to the region below the curve (**in**). Otherwise, it belongs to the region above the curve (**out**). With this function we can choose which part of the triangle we which to render, above the curve or below it. Note that, equation 2 is the implicit form of equation 1, where $D = 3$ and applying triple Knot insertion. An illustration of this process is provided in figure 4. In figure 5, a sample output with different weight values is shown. Also we see the basic difference with [7] where the latter can only render the triangle on the far left.

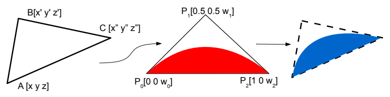


Figure 4: **Left:** Triangle in world coordinates. **Middle:** corresponding mapping in texture space. **Right:** Final Image in screen space.

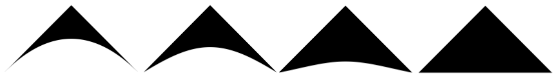


Figure 5: A triangle rendered with decreasing weight values at the off-curve vertex. Left to Right: w_1=1 $\rightarrow$ 0. In comparison, [7] can only render the left-most shape for the given triangle

4. CURVED REGIONS RENDERING

The in-out function of equation 2 does not provide a smoothly curved boundary of the shape, due to aliasing artifacts. In this section, we enhance equation 2, to provide a smooth interpolation between the in and out parts. We enhance this equation, to handle change factor in the (x, y) directions by computing $\bigtriangledown g(x, y)$ according to the chain rule:

$$\bigtriangledown g = \begin{bmatrix} g_y^x - \frac{w_1((w_0 - w_2)u^2 - 2w_0 u + w_0) g_x^x}{(\alpha u^2 + 2\beta u + w_0)^2} \\ \\ g_y^y - \frac{w_1((w_0 - w_2)u^2 - 2w_0 u + w_0) g_x^y}{(\alpha u^2 + 2\beta u + w_0)^2} \end{bmatrix} \tag{3}$$

where

$$\alpha = w_0 - 2w_1 + w_2 \ , \ \beta = w_1 - w_0$$

and g_b^a denotes $\frac{dt}{da}(b)$ the values of the partial derivative of t w.r.t. a in the b direction, and $t = (u, |v|)$ is the absolute value of the texture coordinates at the current location. Absolute values of the texture coordinates are used since we negate v to define that the **out** region is required instead of the **in** region.

The fragment shader of the programmable pipeline supports the computation of functions of the form g_b^a, by local differencing, since GLSL version 1.x. Having the gradient approximation, we compute $e(u, v)$ which resembles the signed distance from the current pixel to the curve.

$$e(u, v) = \frac{1}{2} - sign(v) \frac{f}{||\bigtriangledown g||} \tag{4}$$

The *sign* function is used to provide the ability to render any of the two regions within the triangle (in or out). Hence to render the out region of the triangle, we negate the sign of the v texture coordinate of p_1. Using the value of $e(u, v)$, we classify the fragment according to the equation:

$$class(u, v) = \begin{cases} in & e(u, v) \ge 1 \\ out & e(u, v) \le 0 \\ boundary & \text{otherwise.} \end{cases} \tag{5}$$

In Equation 5, we get a classification of three cases in comparison to the previous two, provided by equation 2. If class(u,v) is determined as **in**, we render the fragment with full color/shade/texture. If the classification is **out**, we discard the fragment or render it with the back color/texture, depending on the rendering technique used. In the **boundary** case, we do a linear interpolation between the back color c_b and the shading color c_f, as defined in equation 6.

$$color = (1 - e(u,v))c_b + e(u,v)c_f \tag{6}$$

Other rendering techniques might need to define only the alpha channel, in that case we use the value of $e(u,v)$ clamped to the region [0 1]. An example of this process is shown in figure 1 where discarded pixels are rendered in red. Figure 6 shows the smoothness provided by the extended approach described above.

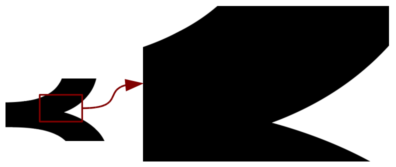


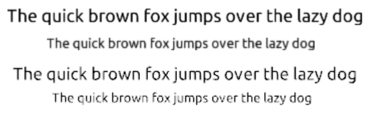
Figure 6: **Left:** A NURBS shape, rendered using our method. **Right:** Zoom in to the area marked in red, showing the smoothness

5. ANTI-ALIASING

In what follows, we will present a View Based Anti Aliasing (VBAA) technique for dealing with skinny small shapes, such as tiny text and condensed P&ID cad drawings. This step is optional since a one pass rendering will provide good results, but to get a crisp overall image an additional rendering pass is required. First, we compute the size d using the following equation:

$$d = r \cdot (M_p \cdot B_w) \tag{7}$$

where r is the radial region of fragments that will affect the final fragment color, M_p is the projection matrix from world coordinates to screen coordinates, and B_w is the bounding box of the shape. We then render the shape into a texture of size d. In the second pass we attach the generated texture to B_w and apply a Gaussian based filter of radius r to produce the final rendered image. We assign horizontal and vertical texels a higher weighting average than diagonal texels. This is to provide the sharp and crisp features of the final shape. A comparison of the output produced by VBAA w.r.t. MSAA is shown in figure 7.



The quick brown fox jumps over the lazy dog
The quick brown fox jumps over the lazy dog
The quick brown fox jumps over the lazy dog
The quick brown fox jumps over the lazy dog

Figure 7: A comparison between rendering using VBAA and hardware MSAA at different sizes. **Top:** A string rendered at two different sizes using the View Based AA. **Bottom:** Same string rendered with 4x MSAA.

6. ACKNOWLEDGMENT

The author is grateful for all the reviewers of this work regarding the algorithm and its application. A special thanks to Sven Gothel and the JogAmp open-source community for the valuable discussions held regarding this work.

7. CONCLUSION

This paper presents a method for a fast and resolution independent rendering of NURBS curves and shapes, without the need for heavy preprocessing. With our method, a user is able to render NURBS shapes in high performance, low memory usage, and high quality anti-aliasing around the curve. We have also shown how our method can be used to render lower order curves (Bezier) such as fonts by setting the weights to one. Finally, since our method does not require heavy computations, it can be used to visualize NURBS shapes on mobile devices. The complete source code of the algorithms presented in this paper is published in JOGL open-source project, part of the JogAmp Community.

8. REFERENCES

[1] Adarsh Krishnamurthy, Rahul Khardekar, and Sara McMains, "Direct evaluation of nurbs curves and surfaces on the gpu," in *Proceedings of the 2007 ACM symposium on Solid and physical modeling*, New York, NY, USA, 2007, SPM '07, pp. 329–334, ACM.

[2] Thomas Warren Sederberg, *Implicit and parametric curves and surfaces for computer aided geometric design*, Ph.D. thesis, West Lafayette, IN, USA, 1983, AAI8400421.

[3] H. Pottmann, S. Leopoldseder, M. Hofer, T. Steiner, and W. Wang, "Industrial geometry: recent advances and applications in cad," *Comput. Aided Des.*, vol. 37, pp. 751–766, June 2005.

[4] Gabriel Taubin, "Distance approximations for rasterizing implicit curves," *ACM Trans. Graph.*, vol. 13, pp. 3–42, January 1994.

[5] Jack Tumblin and Prasun Choudhury, "Bixels: Picture samples with sharp embedded boundaries," in *Rendering Techniques*, Alexander Keller and Henrik Wann Jensen, Eds. 2004, pp. 255–264, Eurographics Association.

[6] Ganesh Ramanarayanan, Kavita Bala, and Bruce Walter, "Feature-based textures," in *Rendering Techniques*, Alexander Keller and Henrik Wann Jensen, Eds. 2004, pp. 265–274, Eurographics Association.

[7] Charles Loop and James Blinn, "Resolution independent curve rendering using programmable graphics hardware," *ACM Trans. Graph.*, vol. 24, no. 3, pp. 1000–1009, 2005.

[8] Sampo Kaasila, "Method and apparatus for moving control points in displaying digital typeface on raster output devices," in *US Patent 5155805*. 1992, Apple Computer, Inc.

[9] Les Piegl and Wayne Tiller, *The NURBS book*, Springer-Verlag, London, UK, 1995.

[10] L. P. Chew, "Constrained delaunay triangulations," in *Proceedings of the third annual symposium on Computational geometry*, New York, NY, USA, 1987, SCG '87, pp. 215–222, ACM.