# Direct Evaluation of NURBS Curves and Surfaces on the GPU

Adarsh Krishnamurthy,* Rahul Khardekar and Sara McMains
Computer Aided Design and Manufacturing Lab
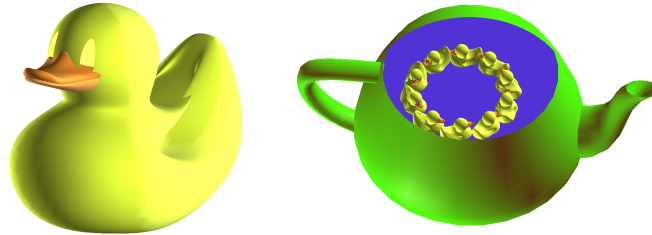University of California, Berkeley

Figure 1: Duck models constructed from trimmed NURBS surfaces evaluated and rendered on the GPU.

## Abstract

This paper presents a new method to evaluate and display trimmed NURBS surfaces using the Graphics Processing Unit (GPU). Trimmed NURBS surfaces, the *de facto* standard in commercial 3D CAD modeling packages, are currently tessellated into triangles before being sent to the graphics card for display since there is no native hardware support for NURBS. Previous GPU-based NURBS display methods relied on first approximating the NURBS patches with lower degree Bezier patches before evaluation. Our method uses a GPU fragment program to evaluate the surface point coordinates of the original NURBS patch directly, from the control points and knot vectors stored as textures in graphics memory. This evaluated surface is trimmed during display using a dynamically generated trim-texture calculated via alpha blending. The implementation incorporates dynamic Level of Detail (LOD) for real-time interaction at different resolutions of the NURBS surfaces. We obtain rendering speeds at least one order of magnitude faster than evaluation using the CPU.

**CR Categories:** I.3.3 [Computer Graphics]: Hardware Architecture—Graphics Processors; I.3.5 [Computer Graphics]: Computational Geometry and Object Modeling—Splines.

**Keywords:** NURBS, GPU, Surface Evaluation, Level of Detail

## 1 Introduction

Non-Uniform Rational B-Splines (NURBS) are the industry standard for the representation of geometry in commercial mechanical Computer Aided Design (CAD) systems. Although NURBS are ubiquitous in the CAD industry, there is currently no built-in

---

*e-mail: {adarsh—rahul—mcmains}@me.berkeley.edu

hardware support for displaying NURBS surfaces. OpenGL provides a software NURBS solution; however, the implementation is not fast enough for evaluating large surfaces interactively, and in our experience it often renders trimmed NURBS surfaces incorrectly. Because surface evaluation is so computationally intensive, the common practice in CAD systems is to preprocess the NURBS surfaces by evaluating and tessellating them into triangles, and then using the standard graphics pipeline to display them. Such a preprocessing technique not only leads to very high memory usage, but also restricts the surface evaluation to a particular Level of Detail (LOD). In this paper we describe a method by which we evaluate and display a trimmed NURBS surface directly on the GPU, without approximating it by simpler surfaces as previous GPU-based algorithms have done. Using the GPU's computational power not only speeds up the surface evaluation significantly but also reduces the CPU memory usage, eliminating the need to calculate and store the tessellation data or simplified surface information that is typically used only for visualization purposes.

NURBS are the most general type of spline curve, encompassing B-splines and Bezier curves as special cases [Piegl 1991]. Although a spline curve may have hundreds of control points, the local support property guarantees that in a B-spline curve of order $k$, the curve evaluation point at any given parameter location is controlled only by the $k$ (parametrically) nearest control points. This simplifies evaluation as well as curve editing and optimization. NURBS offer a way to represent almost arbitrary shapes while maintaining mathematical exactness. NURBS control points can have non-unit weights, which make them rational. Rational curves have the advantage that they can represent conic sections, allowing the exact representation of features with circular cross sections such as rolling ball blends and surfaces of revolution. In addition, the knot vector of the NURBS curve can be non-uniform. Non-uniformity of the knot vector provides one more degree of freedom for the NURBS curves, for example increasing knot multiplicity to change the continuity. The boundary of a CAD model is usually represented by trimmed tensor-product NURBS surface patches.

Graphics processing units (GPUs) have recently evolved into programmable parallel processors capable of performing general-purpose computational tasks. Two programmable units, the Vertex Processing Unit (VPU) and the Fragment Processing Unit (FPU), can execute a user-defined set of instructions, called the vertex program and the fragment program, for each vertex and fragment respectively. Vertex programs can obtain the geometry and attribute

(color, texture coordinates etc.) data stored in the GPU memory via traditional display lists or more recently, Vertex Buffer Objects (VBOs). Fragment programs can access data stored in textures that can have full 32-bit floating-point precision. Usually the output of the FPU goes into a framebuffer, which is a two dimensional block of memory with four attributes at each location. In modern GPUs, the FPU can also output directly to a floating-point texture (render-to-texture) using off-screen render targets called Frame-Buffer Objects (FBOs). This allows the use of the output of a first pass through the rendering pipeline as input texture data for the second pass. FBOs can also be used to render into a Vertex Buffer Object (VBO) so that the output can be used as vertex data for the next rendering pass. Because multiple vertices and pixels are processed in parallel, and operands are four-component vectors, GPUs can achieve much higher computational speeds than conventional CPUs on arithmetically intensive operations.

## 1.1 Related Work

Many early high-quality renderings of curved surfaces used ray tracing [Toth 1985; Nishita et al. 1990; Martin et al. 2000]. Although ray tracing produces very good visual results, it is still not fast enough for real-time interaction with NURBS surfaces.

Before the advent of programmable graphics hardware, parametric surfaces were tessellated before display. The OpenGL version 1.1 implementation renders trimmed NURBS surfaces using the method presented in [Rockwood et al. 1989] for trimmed parametric surfaces. However, in our experience the OpenGL implementation tessellates trimmed NURBS surfaces incorrectly at trim curve concavities and includes spurious triangles in regions where the NURBS surface should be trimmed. Another popular approach was to display the NURBS surface after first converting them to rational Bezier patches or approximating them with non-rational Bezier patches [Kumar and Manocha 1995; Kahlesz et al. 2002], since Bezier evaluation is less computationally demanding. These patches were then triangulated and sent to the graphics card for display.

One method to achieve real-time interaction without tessellation is by using programmable graphics hardware for evaluating and rendering the parametric surface. [Guthe et al. 2005; Guthe et al. 2006] outline a scheme where they use a vertex program to calculate Bezier surfaces that approximate a given NURBS surface. They approximate each NURBS surface with bicubic Bezier patches and then evaluate the Bezier patches on the GPU after the CPU approximation step. They used a LOD system for choosing the appropriate approximation for patch decomposition and the sampling density. Since higher order NURBS surfaces cannot be matched exactly by lower order Bezier surfaces, a disadvantage of this approach is that the final surface may not achieve sufficient accuracy unless it is split into many Bezier patches. This increases the number of Bezier patches by up to two orders of magnitude compared to the number of original NURBS patches in their examples. Moreover, using a vertex program also restricts the number of parameters that can be passed to a vertex program and hence is limited to at most bicubic Bezier patch evaluation. In our approach, we use a fragment program to calculate the surface coordinates.

Fragment programs have formerly been used to evaluate other non-NURBS curved surfaces. Prior work by [Bolz and Schröder 2002; Shiue et al. 2005] focused on using a fragment program to compute the limit points of Catmull-Clark subdivision meshes. These methods can be extended to evaluate uniform B-spline surfaces; the limit surface of a Catmull-Clark subdivision in the absence of extraordinary points is the bi-cubic B-spline surface. However, they cannot be extended to evaluate NURBS because they do not have a subdivision scheme with stationary rules [Sederberg et al. 2003]. Recently, [Loop and Blinn 2006] developed a method in which they solve for the roots of the algebraic equation of a parametric surface using a fragment program. However, due to limitations of the graphics hardware, they are limited to surfaces of degree no more than four (bicubic). Some of these previous fragment-program implementations of surface evaluation slowed down considerably with a large number of surfaces because the evaluated surface coordinates had to be read back from an off-screen pixel buffer using an expensive p-buffer switch for each surface. We get around the p-buffer switch issue by using a frame buffer object, which renders directly to a texture, and a vertex buffer object, which takes this texture as input coordinates for a subsequent rendering pass.

## 1.2 GPU Algorithm

Our algorithm for displaying trimmed NURBS surfaces consists of two operations, one for trimming and the other for surface evaluation. The surface evaluation algorithm is a multi-pass algorithm that uses a fragment program to evaluate the surface point coordinates directly, with no approximations. The trimming operation is based on the approach in [Guthe et al. 2005]. In our implementation, the trimming curves are evaluated and the trim-texture is generated using alpha blending in the graphics card. Finally, while rendering the surface, the actual trimming of the surface is performed on the GPU using another fragment program. Thus, trimming is completely decoupled from surface evaluation. We first describe our NURBS curve evaluation method, the basis for surface evaluation as described next, and then describe the details of trimming.

# 2 NURBS Curve Evaluation

Equation 1 gives the definition of a NURBS curve $C$ as a function of the parameter $u$, where the $P_i$s are the control points and $N_i^p$s are the B-spline basis function of degree $p$ given by Equation 2. The $u_i$s in the definition of the basis functions are the $i^{th}$ components of the knot vector. The $w_i$s are the weights of each control point, making the curve rational. The definition of the basis function is recursive; the basis function of degree $p$ depends on the basis functions of degree $p - 1$. The zero-degree basis function is a step function that is 1 if and only if the parameter value lies in the knot span as defined by Equation 3. Since the NURBS curve can have repeated knot values, the special case of $0/0$ that may arise in one of the terms in Equation 2 is taken to be 0. For concreteness, we consider a NURBS curve of order $k$ with $n$ control points, which has a knot vector of length $n + k$. The knot vector's non-decreasing values are normalized to range from 0 to 1 with at most $k$ repetitions (multiplicity) of knot values for connected NURBS curves. The NURBS curve exists in the knot span $[u_k, u_{n-k}]$.

$$C(u) = \frac{\sum_{i=0}^{n} N_i^p(u) w_i P_i}{\sum_{j=0}^{n} N_j^p(u) w_j} \tag{1}$$

$$N_i^p(u) = \frac{u - u_i}{u_{i+p} - u_i} N_i^{p-1}(u) + \frac{u_{i+p+1} - u}{u_{i+p+1} - u_{i+1}} N_{i+1}^{p-1}(u) \tag{2}$$

$$N_i^0(u) = \begin{cases} 1 & \text{if } u_i \leq u < u_{i+1} \\ 0 & \text{otherwise} \end{cases} \tag{3}$$

Following [Piegl and Tiller 1997], we can break computing the co-ordinates of a point on a NURBS curve given a parameter value $u$ into these three steps:

1. Find the knot span $[u_i, u_{i+1})$ in which $u$ lies, i.e. $u \in [u_i, u_{i+1})$.

2. Compute the corresponding non-zero basis functions $N_{i-p}^p(u), ..., N_i^p(u)$.

3. Multiply the non-zero basis functions with the corresponding control points and sum the results.

The first step, finding the knot span in which $u$ lies, is computed on the CPU. The basis function values corresponding to each control point are then evaluated using a fragment program on the GPU. Finally, the actual curve points are evaluated by multiplying out the values of the basis functions and the corresponding control points, and then adding them together using another fragment program. For clarity, we first describe our procedure for calculating a NURBS curve point without any packing of data or optimization in the following two sections. Details of our data packing and optimizations are presented separately in section 2.3.

## 2.1 NURBS Basis Evaluation

The first step in NURBS curve or surface evaluation is the calculation of the B-spline basis functions, which are dependent only on the knot vector and the parameter value. In order to calculate the basis function values, the knot information has to be transferred to the GPU. For this purpose, a knot array texture is generated on the CPU. This texture of width $2k+1$ and height equal to the number of evaluation points has the value of the parameter $u$ in the first column. The remaining columns have the $2k$ knot values for the evaluation of the corresponding non-zero basis function values for a particular evaluation point. An example of such a knot array is shown in Figure 2, where the values are visualized as a color plot for clarity. This is a sample knot array for evaluating a cubic NURBS curve (order 4) at 100 evaluation points with equally spaced parameter values from 0 to 1. The knot vector for this example is
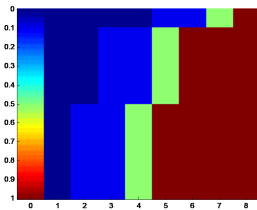$[0.0 \quad 0.0 \quad 0.0 \quad 0.0 \quad 0.1 \quad 0.1 \quad 0.5 \quad 1.0 \quad 1.0 \quad 1.0 \quad 1.0]$.



Figure 2: Knot array of width $2k+1$; knot values to be transferred to the GPU as a texture visualized as a color plot.

Calculation of the basis function is done by constructing the higher order basis functions from the lower order basis functions on the GPU. The first-order (zero-degree) basis function, being the step function, is common for all evaluation points. It is a vector of size $k+1$ and is of the form shown in Equation 4.

$$\left[ \underbrace{0 \quad 0 \quad ... \quad 0}_{k-1} \quad 1 \quad 0 \right] \tag{4}$$

This vector is generated on the CPU; Figure 3(a) shows the generated first-order basis functions for the cubic NURBS curve (order 4)

for 100 evaluation points. The generated first-order basis function is then transferred to the graphics card and stored there as a texture, call it *Tex*1.



(a) $1^{st}$ order

(b) $2^{nd}$ order

(c) $3^{rd}$ order
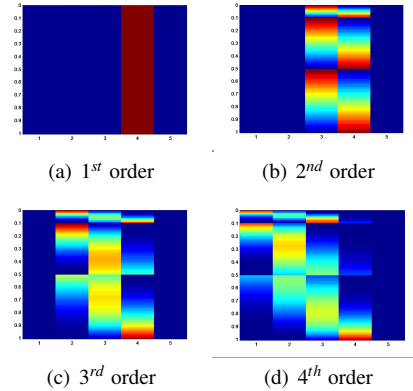
(d) $4^{th}$ order

Figure 3: Intermediate values visualized as a color plot while computing a cubic basis function on the GPU.

The second-order basis function is computed from *Tex*1 and the knot array using a fragment program (see appendix for the fragment program) and is directly rendered to another texture, call it *Tex*2, using the frame buffer object. The third-order basis function is then similarly computed using *Tex*2 as input and rendering to *Tex*1. Thus by alternatively using *Tex*1 and *Tex*2, the higher order basis functions are calculated; a fourth-order basis function is calculated at the end of the third pass. In general, a $k^{th}$-order basis function is computed in $k-1$ passes. Figure 3 shows the output during intermediate passes while computing a fourth-order basis function. This "ping-pong" technique of computing back and forth between two textures is commonly used in GPU programming to deal with cases where the output from an intermediate computation is required at a later stage.

## 2.2 Curve Evaluation

Once the basis function values are calculated, the next step is to multiply these values with their corresponding control points. For this, another array with the corresponding control points for each parameter value to be evaluated is created on the CPU. This control point array is an array of width $k$, with the $x, y, z$ and $w$ values packed in *RGBA* channels. This array is multiplied with the basis function array calculated in the previous step. A fragment program then multiplies all the four channels of the control point array simultaneously with the basis function values. The resulting array is then "reduced" along the width direction to its per-row sum to obtain the actual curve positions. The sequence of steps for calculating the final point coordinates is shown graphically in Figure 4.
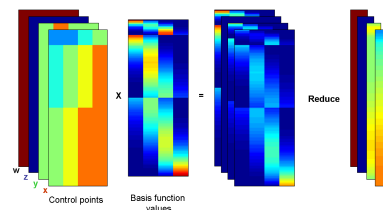


Figure 4: Sequence of steps for curve point evaluation.

## 2.3 Optimization and Packing of Data

GPU calculations are performed simultaneously on all four channels (RGBA); therefore using only one channel for the calculations leads to wasted resources. By packing the data in the knot array in an intelligent manner, we can save storage space as well as speed up the computations. The data can be packed either in the width direction or in the height direction. However, since the width of the array is dependent on the order of the basis function being evaluated, packing it in the width direction will necessitate the use of different fragment programs for different-order curves being evaluated.

The data required for the calculation of the B-spline basis function is completely contained in each row of the knot array. Hence, it will be simpler to pack the data along the height direction with each channel corresponding to different evaluation points as shown in Figure 5. The first entry of each channel in the row specifies the parameter value at which the basis functions are to be evaluated. This kind of packing is also easy to implement since it directly extends from the non-packed version, requiring only very minor changes to the fragment program. It is also not required to have different programs based on the order of the curve being evaluated; the same program generalizes to any order.
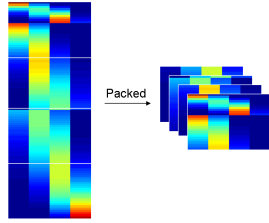


Figure 5: Packing of the knot and basis function data reduces data transfer and GPU computations.

However, there are some disadvantages in packing the data for basis function evaluation. The size of the fragment program for multiplying the control points is larger as the data has to be unpacked and multiplied with four values in a single pass. Moreover, NURBS curves with repeated knot values give rise to the special 0/0 case in their evaluation, which we need to yield a result of 0 rather than the NaN specified by IEEE standards. Although GPUs we have tested return the non-IEEE-compliant 0 that we desire, for greater portability and forward-compatibility we explicitly check for these special cases. Because these have to be handled for each channel separately, it leads to numerous *if* statements in the fragment program increasing its length. Older graphics cards evaluate both branches of if statements and hence they can slow down computation. However, in our experience there was a less than 5% performance drop due to these statements in even in older cards like the ATi Mobility Radeon 9700. Newer graphics cards like the ATi X1900 and the nVIDIA Quadro FX4500 have hardware support (dynamic flow control) for branching and hence this is not a major problem.

## 3 NURBS Surface Evaluation

Recall that the tensor-product NURBS surface definition (Equation 5) is extended directly from that of a NURBS curve. The parameter values $(u,v)$ are the 2D evaluation points; the basis functions $N_i^p$s are the same B-spline basis functions of degree $p$ defined by Equation 2; and the $P_{ij}$s are the NURBS control points defined as a quadrilateral mesh.

$$S(u,v) = \frac{\sum_{i=0}^{n}\sum_{j=0}^{n} N_i^p(u)N_j^p(v)w_{ij}P_{ij}}{\sum_{i=0}^{n}\sum_{j=0}^{n} N_i^p(u)N_j^p(v)w_{ij}} \qquad (5)$$

The NURBS surface is fully defined by a control point mesh and two knot vectors: one for the $u$ parametric direction and the other for the $v$ parametric direction. A general NURBS surface can be of different degree in the $u$ and $v$ directions. So given a control point mesh of size $n \times m$ and of order $k_u$ in the $u$ direction and $k_v$ in the $v$ direction, the knot vector is of size $n+k_u$ in the $u$ direction and $n+k_v$ in the $v$ direction. As in the case of curves, a NURBS surface point is influenced only by a small sub-mesh of control points of size $k_u \times k_v$.

### 3.1 GPU Implementation

Given all the data for a NURBS surface, our surface evaluation algorithm computes the surface point coordinates at parametric coordinates $(u,v)$ in the following manner.

1. Locate the lower left corner of the sub-mesh of control points that influence the evaluation point coordinates.

2. Compute the non-zero basis functions along the two parameter directions.

   (a) Compute the non-zero $u$ basis functions using the $u$ direction knot vector.

   (b) Compute the non-zero $v$ basis functions using the $v$ direction knot vector.

3. Multiply the non-zero basis functions with their corresponding control points from the sub-mesh and sum the results.

The first step of computing the lower left corner control point that influences the current surface point coordinate is equivalent to the first step in the curve evaluation; it is done on the CPU and transferred as 1D texture to the graphics card. The two substeps of the second step are each performed in the same manner as computing the basis functions for curve evaluation explained in section 2.1. Finally, the evaluated basis functions are multiplied with the corresponding control points and added together, as explained in detail below.

Figure 6 represents the surface evaluation process pictorially. We specify the parametric $u$ and $v$ coordinates of the points required to be evaluated in the CPU. We then calculate the basis functions corresponding to these coordinates on the GPU using the basis function evaluation algorithm defined in Section 2.1 and generate the two textures for $u$ and $v$ having the basis function values at the required parameter coordinates.
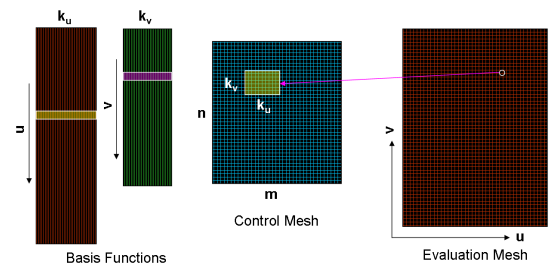


Figure 6: Graphical representation of the surface evaluation algorithm.

Once the basis functions are evaluated, we again alternate (ping-pong) between output textures to evaluate the final surface coordinates. We store the control point data in a texture of size $n \times m$ in the GPU memory. We also have a texture of size equal to the evaluation mesh, call it $tex1$, which is initialized to zero. Given a particular $u$ and $v$ coordinate, we look up the coordinates of the control point that influences the current evaluation point using the index values stored in the 1D textures calculated in step 1. We then multiply this control point with its corresponding $u$ and $v$ basis function values and add it to $tex1$ using a fragment program. This fragment program directly renders the multiplied result to another texture, call it $tex2$. In the next pass, the newly multiplied values of this pass are added to $tex2$ and rendered directly to $tex1$. Thus, the final curve point is evaluated in $k_u \times k_v$ passes; for example, a bi-cubic NURBS surface point is evaluated in 16 passes.

## 3.2 Continuous LOD

The NURBS patches that make up a particular model or a scene are usually of different sizes and at different magnification levels. In such cases, it would be inefficient to evaluate all the surfaces at the same level of detail. Therefore, we use different evaluation grids for different surfaces based on the size of the surface and the distance of the surface from the eye point. Older graphic cards were optimized to only work with square power-of-2 textures. Hence, the transitions between the different LODs are not smooth leading to popping artifacts between them. Furthermore, it was not possible to have different number of evaluation points along the $u$ and $v$ directions. However, newer graphic cards support rectangular textures of any size. Thus, for the different LODs, the number of evaluation points change continuously from the minimum to the maximum value in our implementation. In addition, the number of evaluation points are different for the $u$ and $v$ directions. This leads to a better rendering of dynamic scenes encountered in interactive environments like solid modeling.

We compute the required height and width of the evaluation mesh by finding the distance of the object from the eye point as well as the size of the object. Then the connectivity of the points is generated on the CPU using the selected size. We make use of the fact that the connectivity of a 2D mesh in the parametric domain is the same as the connectivity of the final NURBS surface. This index information is sent to the graphics card and the surface is rendered by using the corresponding point coordinate data taken directly from a texture using a vertex buffer object. This way we eliminate a redundant and costly operation of reading back the evaluated point coordinates from the GPU and then sending them back as vertex coordinates.

## 4 Trimming

For efficient rendering of a trimmed NURBS surface, the surface evaluation should be decoupled from trimming. Instead trimming can be performed with the help of texture mapping using a trim-texture, a trimming technique first applied to trimmed spline surfaces by [Guthe et al. 2005]. The trim-texture is generated by evaluating and rendering the trim curves in the 2D parametric domain. If the trim curves are described by splines, they can be evaluated and converted to piecewise linear segments. As described in [Woo et al. 2004], arbitrary concave polygons (possibly even including holes) do not need to be tessellated for rendering. Instead, triangles connecting a common origin to each polygon edge in turn are rasterized, but only those regions that are filled an odd number of times

are finally rendered. The above algorithm can be implemented either by using the stencil buffer or by alpha blending. Using the stencil buffer is sufficient to trim surfaces that are parallel to the view plane; implementation details for using the stencil buffer are given in [Woo et al. 2004]. However, we use an alternate implementation based on the alpha blending functionality of graphics cards to generate the trim-texture because the trimmed surfaces may be arbitrarily oriented or curved.

The trim-texture is then used to mask parts of the surface using a fragment program during the rendering pass. Even though the trim-texture has alpha values that can be mapped directly to the surface by using alpha blending, this may lead to incorrect results. Unless all the objects are rendered in back-to-front order, the blending will not be correct; the objects behind discarded trim portions will not be rendered. The problem becomes even more pronounced in the case of curved surfaces, where the surface itself may be self-occluding. In this case, since the order in which the fragments are processed by the graphics card is not defined, the final surface will be rendered incorrectly and can even have artifacts similar to self-shadowing. To overcome this problem, only the parts of the surface that lie outside the trim curves are rendered.

## 5 Results

We tested our evaluation method on the different GPU platforms listed in Table 1.

| GPU | VRAM | CPU | RAM |
|---|---|---|---|
| ATi X1900 | 512 MB | 2.8 GHz | 512 MB |
| nVIDIA QuadroFX 4500 | 512 MB | 3.0 GHz | 2048 MB |
| nVIDIA QuadroFX 3000 | 256 MB | 1.88 GHz | 1024 MB |
| nVIDIA GeForceFX 6800Go | 256 MB | 1.6 GHz | 512 MB |

Table 1: Different GPU platforms tested.

Figure 7 compares the evaluation timing alone of a single bi-cubic NURBS patch defined by 144 control points when increasing the density of the evaluation grid. The GPU-based evaluation is faster than the CPU-based evaluation by a factor of about 50 when evaluated at a large numbers of evaluation points. However, the GPU evaluation has more overhead for very small patches and hence is not suitable for evaluating surfaces having less than 16 evaluation points. The nVIDIA QuadroFX 3000 is an older graphics card and uses AGP8x bus architecture. Hence, the data bandwidth is not as high as the other PCI-e graphics cards tested. As a result, the timings are somewhat slower but still about 10 times faster than on a CPU. The high end PCI-e 16x graphics cards from both ATi and nVIDIA produced almost identical results.

The duck model shown in Figure 1 consists of three NURBS surfaces with both non-uniform knots and non-unity weights for the control points. One of the three surfaces in the model is also trimmed. Figure 1 is rendered using an evaluation grid of $64 \times 64$ points for each surface on a window of size $1280 \times 1024$. Note that the trimmed yellow patch representing the duck's body fills most of this window, but has no obvious tessellation artifacts with this sampling density. The model can be interactively displayed with varying LODs without re-sending the data to the GPU repeatedly. Similarly, some changes to the model will necessitate transferring only the control points to the GPU.

Figure 8 shows the frame rates for an animated scene containing many such ducks swimming in a (tessellated) teapot, similar to Figure 1, comparing our GPU implementation to the OpenGL implementation. The scene is again rendered in a window of size
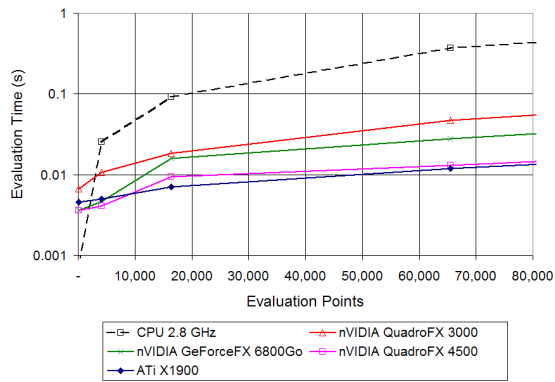
Figure 7: Comparison of evaluation timings for a bi-cubic NURBS surface with increasing evaluation points. Note logarithmic scale for the y-axis.

$1280 \times 1024$; the individual NURBS surfaces, being smaller than the full screen area, were evaluated on a $16 \times 16$ grid of evaluation points. One-third of the NURBS surfaces were non-trivially trimmed (the OpenGL timings do not include trimming). As expected, the frame rate decreases with the increase in the number of surfaces. However, the decrease in frame rate is not linear in the number of surfaces. This may be due to the extra overhead of transferring the control points data for a large number of surfaces to the graphics card and some overhead in switching between the VBO of different surfaces. Even though trimming was not performed while obtaining the OpenGL-rendered timings, its frame rates are unacceptably slow for more than about 100 NURBS surfaces, consistently 40-50 times slower than our GPU-based implementation.
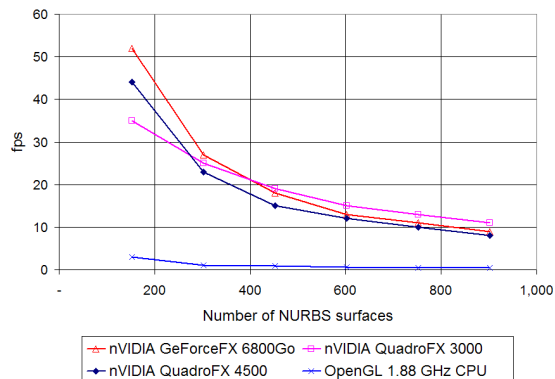


Figure 8: Comparison of frame rates with different nVIDIA graphics cards. One-third of the total NURBS surfaces are non-trivially trimmed.

# 6 Summary and Conclusions

We have presented a new method to evaluate and display trimmed NURBS surfaces on the GPU. The method shows great promise for real-time interaction with exact NURBS models, as seen from the framerates we achieved even on older graphics cards. Our algorithm evaluates the NURBS surface point coordinates directly, without resorting to approximations that can dramatically increase the number of surfaces that need to be calculated, stored, and dis-

played. The evaluation timings show at least an order-of-magnitude improvement over evaluation on the CPU for large inputs, and a similar improvement in overall framerate compared to the OpenGL implementation. For interactive display of a large number of trimmed NURBS surfaces, we have demonstrated that GPU-based evaluation of the exact surfaces is a viable option.

# References

BOLZ, J., AND SCHRÖDER, P. 2002. Rapid evaluation of Catmull-Clark subdivision surfaces. In *Web3D 2002*, 11–17.

GUTHE, M., BALÁZS, A., AND KLEIN, R. 2005. GPU-based trimming and tessellation of NURBS and T-spline surfaces. *ACM Trans. Graph. 24*, 3, 1016–1023.

GUTHE, M., BALÁZS, A., AND KLEIN, R. 2006. GPU-based appearance preserving trimmed NURBS rendering. *Journal of WSCG 14*.

KAHLESZ, F., BALÁZS, A., AND KLEIN, R. 2002. Multiresolution rendering by sewing trimmed NURBS surfaces. In *SMA '02: ACM symposium on Solid modeling and applications*, 281–288.

KUMAR, S., AND MANOCHA, D. 1995. Efficient rendering of trimmed NURBS surfaces. *Computer-aided Design 27*, 7, 509–521.

LOOP, C., AND BLINN, J. 2006. Real-time GPU rendering of piecewise algebraic surfaces. In *ACM SIGGRAPH 2006*, 664–670.

MARTIN, W., COHEN, E., FISH, R., AND SHIRLEY, P. 2000. Practical ray tracing of trimmed NURBS surfaces. *Journal of Graphics Tools: JGT 5*, 1, 27–52.

NISHITA, T., SEDERBERG, T. W., AND KAKIMOTO, M. 1990. Ray tracing trimmed rational surface patches. In *ACM SIGGRAPH 90*, 337–345.

PIEGL, L. A., AND TILLER, W. 1997. *The NURBS Book, Version 1.2*, second ed. Springer.

PIEGL, L. 1991. On NURBS: a survey. *IEEE Comput. Graph. Appl. 11*, 1, 55–71.

ROCKWOOD, A., HEATON, K., AND DAVIS, T. 1989. Real-time rendering of trimmed surfaces. In *ACM SIGGRAPH 89*, 107–116.

SEDERBERG, T. W., ZHENG, J., BAKENOV, A., AND NASRI, A. 2003. T-Splines and T-NURCCs. *ACM Trans. Graph. 22*, 3, 477–484.

SHIUE, L.-J., JONES, I., AND PETERS, J. 2005. A realtime GPU subdivision kernel. *ACM Trans. Graph. 24*, 3, 1010–1015.

TOTH, D. L. 1985. On ray tracing parametric surfaces. In *ACM SIGGRAPH 85*, 171–179.

WOO, M., NEIDER, J., DAVIS, T., AND SHREINER, D. 2004. *OpenGL(R) Programming Guide, Version 1.4*, fourth ed. Addison-Wesley, ch. Drawing Filled Concave Polygons Using the Stencil Buffer, 600–601.