Computer-Aided Design 41 (2009) 971–980

Contents lists available at ScienceDirect

Computer-Aided Design

journal homepage: www.elsevier.com/locate/cad

Optimized GPU evaluation of arbitrary degree NURBS curves and surfaces

Adarsh Krishnamurthy *, Rahul Khardekar, Sara McMains

Computer Aided Design and Manufacturing Lab, University of California, Berkeley, United States

ARTICLE INFO

Article history:
Received 16 July 2008
Accepted 13 June 2009

Keywords:
NURBS
GPU
Surface evaluation
Level of detail

ABSTRACT

This paper presents a new unified and optimized method for evaluating and displaying trimmed NURBS surfaces using the Graphics Processing Unit (GPU). Trimmed NURBS surfaces, the de facto standard in commercial mechanical CAD modeling packages, are currently being tessellated into triangles before being sent to the graphics card for display since there is no native hardware support for NURBS. Other GPU-based NURBS evaluation and display methods either approximated the NURBS patches with lower degree patches or relied on specific hard-coded programs for evaluating NURBS surfaces of different degrees. Our method uses a unified GPU fragment program to evaluate the surface point coordinates of any arbitrary degree 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 display also incorporates dynamic Level of Detail (LOD) for real-time interaction at different resolutions of the NURBS surfaces. Different data representations and access patterns are compared for efficiency and the optimized evaluation method is chosen. Our GPU evaluation and rendering speeds are more than 40 times faster than evaluation using the CPU.

1. Introduction

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

The use of a preprocessing technique not only leads to very high memory usage, but also restricts the surface evaluation to a particular Level of Detail (LOD). Hence, a highly enlarged view of the surface may not be tessellated sufficiently, whereas a distant view may render an excessive number of triangles. In this paper, we describe a method by which we evaluate and display a trimmed NURBS surface directly, without approximating it by simpler surfaces, using a programmable graphics card. The usage of the GPU's computational power not only speeds up the surface evaluation significantly but also reduces the CPU memory usage, eliminating the need for calculating and storing the tessellation data or simplified surface information that is typically used only for visualization purposes.

Previous GPU methods [1,2] focused mainly on rendering NURBS surfaces rather than exact evaluation. Hence, they approximated a higher degree NURBS surface by lower degree Bezier surfaces that closely resemble the original surface based on pixel location error while rendering. Even though such approximations are good enough for rendering, they cannot be extended to a general-purpose NURBS evaluator capable of handling arbitrary degree NURBS surfaces. We introduced a unified method to evaluate arbitrary degree NURBS surfaces on the GPU without making any approximations [3]. The contemporaneous work by Kanai [4] for evaluating NURBS surfaces also did not use any approximations, but required different GPU programs for evaluating NURBS surfaces of different degrees. This makes the implementation of their system tedious, since specific new programs have to be written for surfaces of different degrees. Moreover, since standard CAD models can be made of surfaces of widely varying degrees, with surfaces up to degree 100 occurring in many complex models, a unified NURBS evaluation algorithm will be a more practical solution.

In this paper we describe our unified NURBS evaluation and rendering method, expanded from the original conference presentation [3]. The main contributions of our approach include:

- A GPU method for evaluating arbitrary degree NURBS surfaces with an arbitrary number of control points and knots with

* Corresponding author. Tel.: +1 510 590 7325.
E-mail addresses: adarsh@me.berkeley.edu (A. Krishnamurthy), rahul@me.berkeley.edu (R. Khardekar), mcmains@me.berkeley.edu (S. McMains).

0010-4485/$ – see front matter © 2009 Elsevier Ltd. All rights reserved.
doi:10.1016/j.cad.2009.06.015

the same unified fragment program. Our method uses the GPU to evaluate a grid of points on the NURBS surface that can be directly used for rendering as well as for further modeling operations. Our method is easily extensible to evaluate derivatives and normals of the NURBS surface.
- Backward-compatible algorithms that make use of standard OpenGL extensions or features that are available even in cards that are more than 5 years old, while still taking advantage of the improved performance on newer cards.
- Different implementations of the evaluation algorithm that use different memory access patterns and data packing on the GPU. We choose the optimum evaluation method based on the performance of these different implementations.
- A direct method to render trimmed NURBS surfaces by interpreting the points already evaluated as vertices. The rendering algorithm is capable of dynamic continuous LOD based on the size and location of the surface with respect to the view point.

2. Background and related work

2.1. Programmable GPUs

Graphics processing units (GPUs) have recently evolved into programmable parallel processors capable of performing general-purpose computational tasks [5,6]. We make use of two programmable units on the GPU, the Vertex Processing Unit (VPU) and the Fragment Processing Unit (FPU), which can execute a user-defined set of instructions, called the vertex program and the fragment program, for each vertex and fragment respectively, in the place of a fixed sequence of geometric transformations, lighting operations (per-vertex operations), and texturing operations (per-fragment operations). 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). Geometric primitives (triangles generally) assembled from the vertex data then get rasterized into fragments (potential pixels) that pass through the FPU. Vertex and 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 frame buffer, which is a 2D 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.

2.2. NURBS evaluation techniques

Many early high-quality renderings of curved surfaces used ray tracing. Toth [7] and Nishita et al. [8] perform ray tracing on parametric and rational surfaces by solving for the ray-surface intersection point using numerical methods. Martin et al. [9] gives a complete algorithm for ray tracing trimmed NURBS. Pabst et al. [10] used ray casting on the GPU to render trimmed NURBS surfaces.

To take advantage of graphics hardware, parametric surfaces tend to be tessellated before display. Much work on trimmed NURBS focuses on the trimming aspect. The OpenGL version 1.1 implementation renders trimmed NURBS surfaces using the method presented by Rockwood et al. [11] for trimmed parametric surfaces, which divides the parametric domain into patches based on the trim curves. These patches are then tessellated in the 2D domain and then evaluated to find the surface point coordinates. However, in our experience the OpenGL implementation tessellates trimmed NURBS surfaces incorrectly at trim curve concavities. In addition, being a CPU evaluator, it is not fast enough to render large numbers of trimmed NURBS surfaces at interactive rates.

Previous work such as [12–14] displayed NURBS after first converting them to Bezier patches and converting the trimming curves to Bezier segments, since Bezier evaluation is less computationally demanding. These patches were then triangulated and sent to the graphics card for display. Guthe et al. [1,2] approximate each NURBS surface with lower degree Bezier patches, but they then evaluate the Bezier patches on the GPU after the CPU approximation step. They also introduced a LOD system for choosing the appropriate approximation patch decomposition and the sampling density. Since in general no Bezier surface of lower degree can exactly match an arbitrary degree NURBS surface, a disadvantage of this approach is that the final surface may not achieve sufficient accuracy unless it is split into many Bezier patches, increasing the number of patches by up to two orders of magnitude in their examples.

Subdivision surfaces, which have largely replaced tensor-product patches in entertainment applications where mathematical exactness is not required, have also been directly evaluated on the GPU. Prior work by Bolz and Schröder [15] and Shiue et al. [16] 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 [17,18]. Loop and Blinn [19] used the GPU to render piecewise algebraic surfaces of lower degrees. However, it is difficult to extend the method to evaluate arbitrary degree NURBS surfaces.

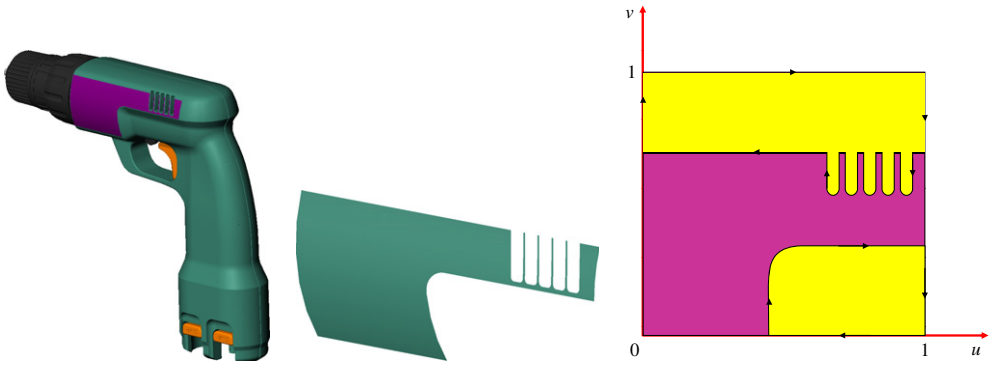
The fragment-program implementations of surface evaluation of subdivisions were not fast enough for real-time interaction with a large number of surfaces because the evaluated surface coordinates had to be read back from an off-screen pixel buffer using an expensive p-buffer switch for each surface. Guthe et al. [1] overcome this issue by using a vertex program, but their method is not as flexible because the number of parameters that can be passed to a vertex program is quite limited, and vertex texture fetches are possible only in the latest graphic cards. Thus, they approximated the original input by a hierarchy of bi-cubic Bezier patches to limit the amount of data that needed to be transferred per patch. In our approach, we use a fragment program but 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.

Recently, Kanai [4] developed a fragment-program based NURBS evaluation that closely resembles our method. However, their implementation required different fragment programs for surfaces of different degrees. While this method is theoretically capable of evaluating any NURBS surface, its implementation becomes tedious since different fragment programs have to be written specifically for each possible degree of a NURBS surface that may be present in a model. Hence a unified evaluation method that can be used to evaluate arbitrary degree NURBS surfaces is preferred.

2.3. NURBS curve and surface definitions

In this section, we briefly review the mathematical notation used for defining NURBS curves and surfaces, adapted from Piegl

Fig. 1. NURBS models constructed from trimmed NURBS surfaces evaluated and rendered on the GPU.



(a) Cordless drill model. (b) Trimmed NURBS surface. (c) Trim curves in the parametric domain.

Fig. 2. Cordless drill modeled using trimmed NURBS surfaces.

and Tiller [20]. Eq. (1) gives the definition of a NURBS curve C as a function of the parameter u, where the P_is are the control points and N_i^ps are the B-spline basis function of degree p given by Eq. (2). Since the NURBS curve can have repeated knot values, the special case of 0/0 that may arise in either of the terms in Eq. (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 $n + k$, in all the examples in this paper. 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.

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

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

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

Recall that the tensor-product NURBS surface definition (Eq. (4)) is extended directly from that of a NURBS curve. The parameter values (u, v) are the 2D evaluation points; the basis functions N_i^ps are the same B-spline basis functions of degree p defined by Eq. (2); and the P_{ij}s are the NURBS control points defined as a quadrilateral mesh. The NURBS surface is fully defined by a control point mesh and the two independent arbitrary degree u and v parametric direction knot vectors. As in the case of curves, a NURBS surface point is influenced only by a small sub-mesh of control points of size $k_u \times k_v$.

$$S(u, v) = \frac{\sum_{i=0}^{n} \sum_{j=0}^{m} N_i^p(u) N_j^q(v) w_{ij} P_{ij}}{\sum_{i=0}^{n} \sum_{j=0}^{m} N_i^p(u) N_j^q(v) w_{ij}}. \tag{4}$$

The boundary of a CAD model is usually represented by tensor-product NURBS surface patches. These surfaces are rectangular sheets; therefore they are not very flexible, especially when it comes to representing surfaces that are not rectangular or those with holes or complex local geometries that arise due to Boolean operations. Therefore, many NURBS patches are trimmed, discarding a part of the surface portion defined in the parametric domain. An example of a trimmed NURBS surface in a CAD model is shown in Fig. 2. The trimming information is defined in the 2D parametric domain of the surface (Fig. 2(c)). Typically, trim curves are represented as directed closed loops; the direction of the loop determines which side of the trim curve to cut away. There can also be multiple loops per surface, one defining the boundary and others defining interior holes, or even holes within holes. Following OpenGL, we have at least one trim curve that bounds the valid surface region for every surface in order to have a consistent representation.

3. GPU evaluation and rendering algorithm

Our NURBS evaluation algorithm consists of two steps: the first step is to evaluate the NURBS basis functions and the second step is to multiply these basis function values with the control points to get the curve or surface point coordinates. This is a multi-pass algorithm that uses fragment programs to evaluate the surface point coordinates without any approximations. For rendering trimmed NURBS surfaces, we make use of our evaluation algorithm to evaluate points on the surface and then use the

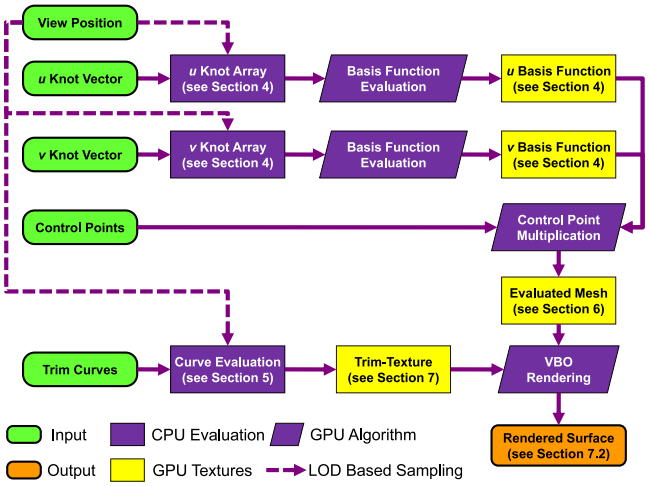


Fig. 3. Algorithm for rendering trimmed NURBS surface.

GPU to trim the unwanted parts of the surface while rendering. The trimming operation is directly adapted from the approach by Guthe et al. [1]. 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. The flow of the different operations, some of which are performed on the CPU, are shown in Fig. 3.

To obtain optimum performance, we distribute the different operations to be performed either on the CPU or on the GPU. Inherently serial operations, such as calculation of the knot array, are better suited to be performed on the CPU. Operations such as basis function evaluation and NURBS surface point evaluation are numerically intensive operations well suited for the better floating point performance of the GPUs. Hence we parallelize these operations and perform them on the GPU. However, even though curve evaluation can be performed on the GPU, the performance gains, if any, were small (see Section 5.3). Hence we perform curve evaluation on the CPU itself.

4. NURBS basis function 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. We need to transfer the information corresponding to the knot values to the GPU in order to calculate the basis function values. For this purpose, we generate a knot array texture on the CPU. The algorithm by Kanai [4] on the other hand, performs this operation using binary-search on the GPU. We perform this on the CPU since the operation does not involve numerically intensive calculations; performing it on the CPU will make the algorithm balanced in terms of CPU/GPU workload.

The knot array texture has the value of the parameter u in the first column; it has dimensions of width $2k + 1$ and height equal to the number of evaluation points. 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 Fig. 4, where the values are visualized as a color plot for clarity. This is a sample knot array for evaluating a cubic NURBS curve at 100 evaluation points with equally spaced parameter values from 0 to 1. The knot vector for this example is

[0.0 0.0 0.0 0.0 0.1 0.1 0.5 1.0 1.0 1.0 1.0].

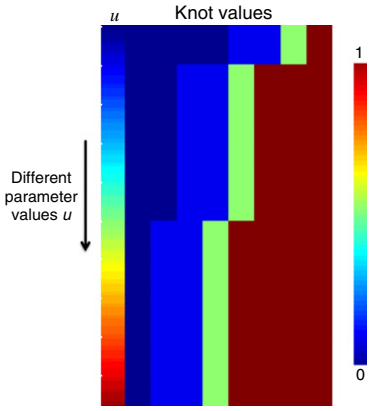


Fig. 4. Knot array; knot values to be transferred to the GPU as a texture visualized as a color plot. (For interpretation of the references to colour in this figure legend, the reader is referred to the web version of this article.)

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 Eq. (5).

$$\left[\underbrace{0 \quad 0 \quad \ldots \quad 0}_{k-1} \quad 1 \quad 0\right]. \tag{5}$$

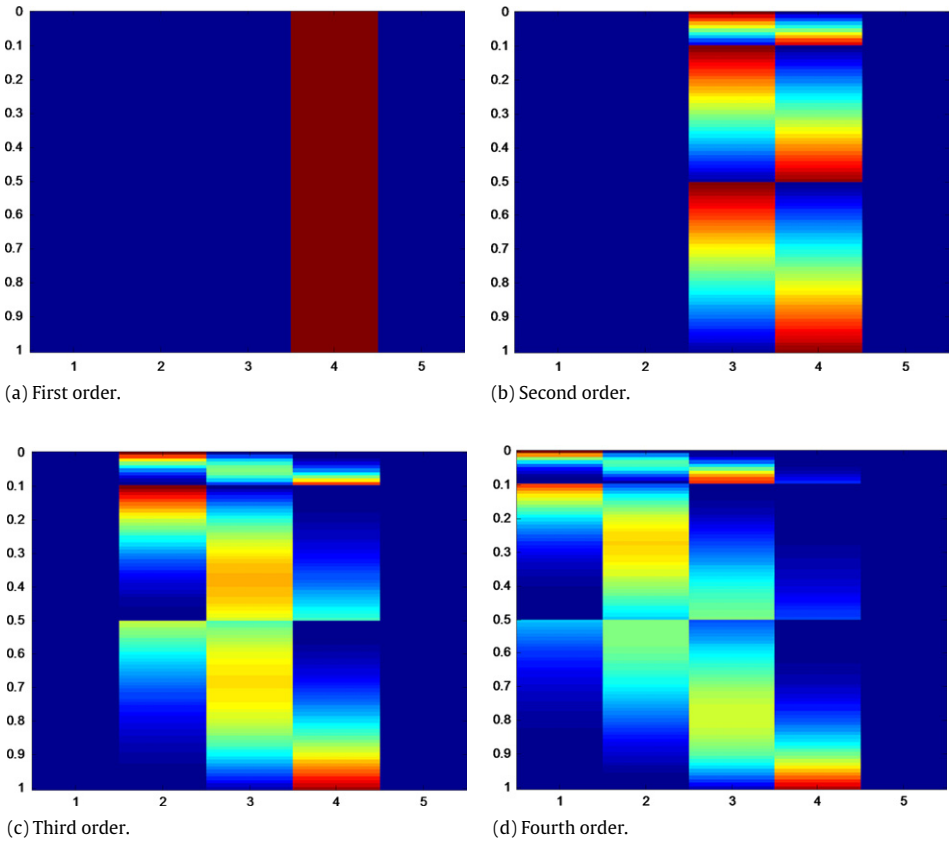
This vector is generated on the CPU; Fig. 5(a) shows the generated first-order basis functions for the cubic NURBS curve (order 4) for 100 evaluation points. The generated first-order basis function is then transferred to the graphics card and stored there as a texture, call it *tex*1.

The second-order basis function is computed from *tex*1 and the knot array using a 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 back 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 kth-order basis function is computed in $k - 1$ passes. Fig. 5 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. The last column is always 0 during the evaluation; however we still store the values in the texture to prevent introducing a branch in the code for evaluation. The additional 0 column unifies the code for evaluation since the access pattern is the same for evaluating all higher-order basis functions.

5. Curve evaluation

Following Piegl and Tiller [20], we can break computing the coordinates 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 function values $N^p_{i-p}(u), \ldots, N^p_i(u)$.
3. Multiply the non-zero basis function values with the corresponding control points and sum the results.



(a) First order. (b) Second order.

(c) Third order. (d) Fourth order.

Fig. 5. Intermediate values visualized as a color plot while computing a cubic basis function on the GPU. (For interpretation of the references to colour in this figure legend, the reader is referred to the web version of this article.)

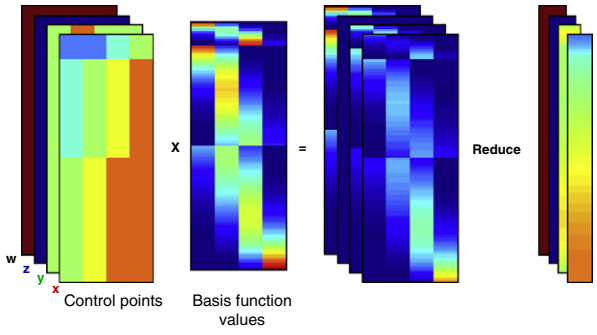


Fig. 6. Sequence of steps for curve point evaluation.

The first step, finding the knot span in which u lies, is computed on the CPU; this operation is essentially performed while generating the knot array 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 section. Details of our data packing and optimizations are presented separately in Section 5.2.

5.1. Basic algorithm

We first compute the basis function values using the GPU evaluation method described in Section 4. 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 stored in the RGBA channels. This array is multiplied with the basis function array calculated in the previous step. A fragment program 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 using a different fragment program. The sequence of steps for calculating the final point coordinates is shown graphically in Fig. 6.

5.2. Optimization and packing of data

The previous section described our method for curve evaluation without any packing of data or optimization. We now describe two techniques that reduce the evaluation time.

GPU calculations are performed simultaneously on all four channels (RGBA); therefore using only one channel for the calculations leads to wasted resources. Packing of data refers to using the four channels to store and process the data instead of using just a single channel. 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 degrees of the curves being evaluated. This will make the implementation tedious because the program for the packed version cannot be directly extended from the non-packed version. It is also impractical

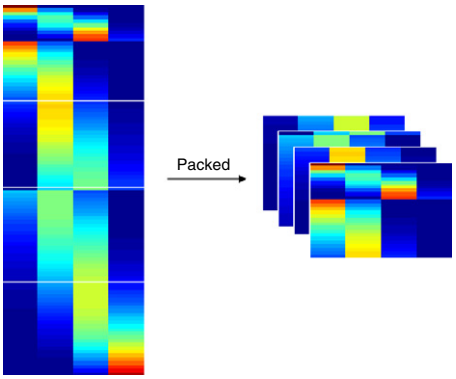


Fig. 7. Packing of the knot and basis function data reduces data transfer and GPU computations.

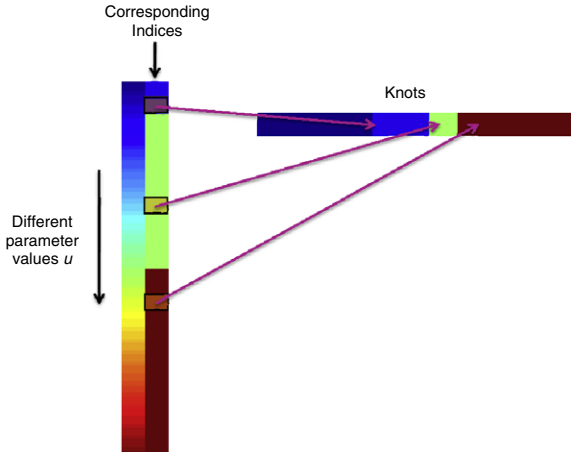


Fig. 8. Using an index array to prevent data duplication.

because different programs have to be developed, one each for each different degree of curve 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 Fig. 7. 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. In addition, the data access from lower degree basis function to evaluate higher degree basis functions in the fragment program remain the same for a particular evaluation point. 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.

However, there is a disadvantage in packing the data for basis function evaluation. 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 many 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. Moreover, since the current generation GPUs are moving towards IEEE-compliance, they will return a NaN value. Since these 0/0 cases have to be separately handled for each channel, 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 the computation. However, the performance drop due to these statements in our

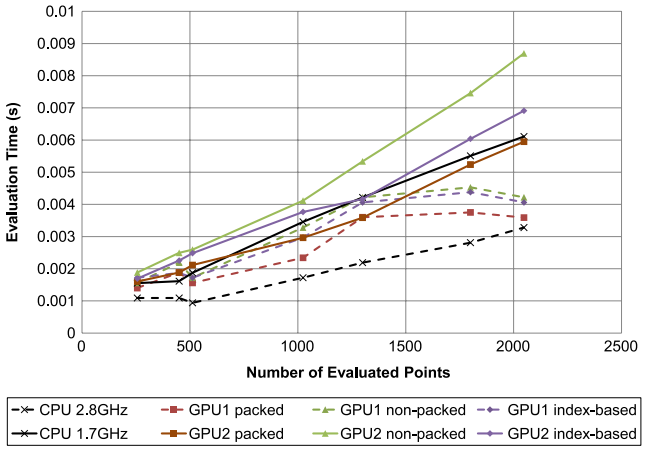


Fig. 9. Time for evaluating a cubic NURBS curve on two different GPUs.

implementation is negligible if any. The difference in the timings even in older cards like the ATi Mobility Radeon 9700 is less than 5%. Newer graphics cards have hardware support (dynamic flow control) for branching and hence this is not a major problem.

We now describe the second, alternative optimization technique we implemented. In the evaluation of the basis function in the example given in the previous section, many knot values were repeatedly used. For example, the knot values required for the computation of the first 10 parameter values shown in Fig. 4 use the same knot values. One method to reduce the amount of data transfer in such cases is to use an index array, which contains indices pointing to the knot values needed for the basis function evaluation. The knot values are stored separately in another array and are transferred directly from the CPU to the GPU. The knot array will then only contain the parameter value and the index of the first element in the knot vector required for the evaluation of the basis functions (Fig. 8).

Using an index array also has its advantages and disadvantages. There is an obvious reduction in data transfer. On the other hand, the GPU architecture is not optimized for such texture indirections or nested texture fetches. The cache is optimized to retrieve data quickly from nearby memory locations; the cache misses are presumably the reason that too many texture indirections can significantly slow performance by introducing too much latency (latency that can no longer be hidden by the parallel nature of fragment processing). In addition, the indexed data cannot be packed anymore because the different channels will point to different knot positions. Hence even if the data is packed, it will require four texture fetches that offset the advantage gained by packing. Therefore, we cannot combine our two techniques.

5.3. Curve evaluation timings

Using the above variations of the GPU algorithm, we timed the evaluation of NURBS curves on different GPUs. Timings were done on four different implementations: CPU, GPU packed, GPU non-packed, and GPU index-based. The non-packed implementation is the regular implementation without any packing or indexing as described in Section 5.1.

Fig. 9 shows the curve evaluation timings for a cubic NURBS curve with different numbers of evaluation points evaluated on ATi Radeon X1900 (GPU1) and ATi Mobility Radeon 9700 (GPU2) graphics cards. The CPUs used for the evaluation were Intel Pentium-4 2.8 GHz and Intel Centrino 1.7 GHz processors respectively. As expected, the evaluation time increases roughly linearly with the number of points evaluated. It can be seen that the packed method is a bit faster than the 1.7 GHz CPU

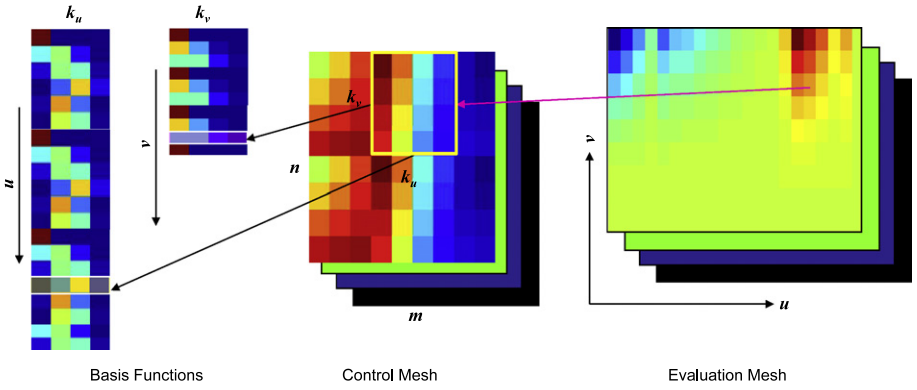


Fig. 10. Graphical representation of the surface evaluation algorithm.

evaluation. However, the other methods are slower than the CPU method on both platforms, either due to the amount of data transferred in the case of the unpacked implementation or due to the texture indirection in the case of the index-based implementation. Evaluation timings on other GPUs also followed the same qualitative trend, with the packed version always the fastest of the GPU methods.

From these results for 2D NURBS curves, it is not immediately clear that a GPU implementation for NURBS surface evaluation will be enough of an improvement over CPU evaluation to justify the development effort. However, in the case of surface evaluation, with its higher arithmetic intensity, the GPU win over CPU is far more pronounced, as described in the later results section. Since we found the GPU packed method of evaluating the basis functions to be the fastest of the three different techniques we developed, we use this method in the surface evaluation algorithm. Since the surface control points used for surface evaluation are already four-component vectors (XYZW), additional data packing is not required for surface evaluation.

6. NURBS surface evaluation

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 a 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 4. Finally, the evaluated basis functions are multiplied with the corresponding control points and added together, as explained in detail below.

Fig. 10 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 4 and generate the two textures for u and v having the basis function values at the required parameter coordinates. We implemented the packed version of the basis function evaluation algorithm because it was the fastest among the different methods discussed in Section 5.2.

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 *tex*1, 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 the corresponding pixel in *tex*1 using a fragment program. This fragment program directly renders the multiplied result to another texture, call it *tex*2. In the next pass, the newly multiplied values of this pass are added to *tex*2 and rendered directly back to *tex*1. 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. In our current implementation, since we evaluate each surface separately, it does not matter if the processed surfaces have different degrees.

6.1. Dynamic 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 efficient to have different numbers 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 such as solid modeling. Fig. 11 shows a duck model rendered at different zoom levels. The LOD varies continuously between the different levels, resulting in smooth transitions.

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

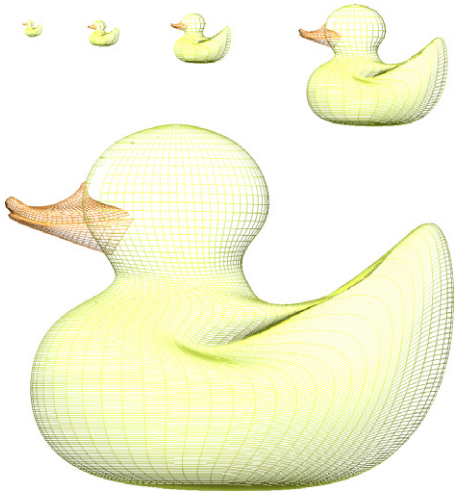


Fig. 11. Dynamic LOD: Duck rendered at different resolutions based on the required LOD.

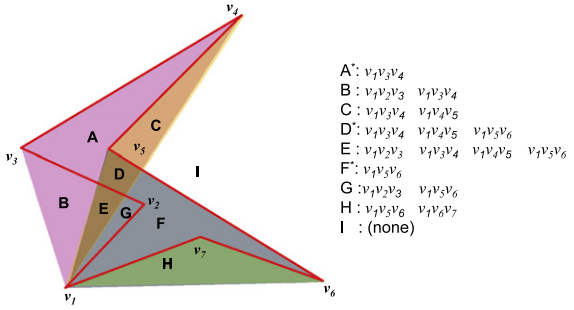


Fig. 12. Adapted from the OpenGL Programming Guide: Example of a trim-texture. Only the starred regions that are rendered an odd number of times are finally displayed.

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.

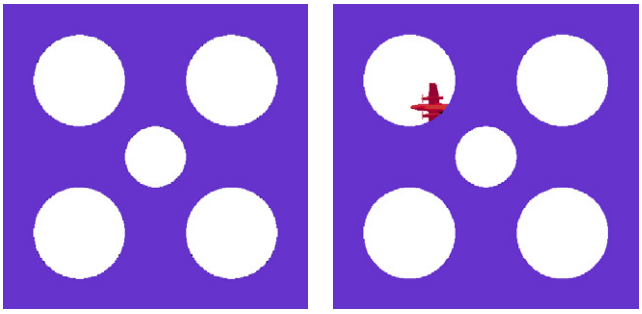
7. Trimming

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. [1].

The trim-texture is generated by evaluating and rendering the trim curves in the 2D parametric domain. Even though NURBS curves can theoretically be used for trim curves, most of the trim curves in practice are piecewise linear segments. This is because a space curve on a 3D NURBS surface is usually approximated by linear segments in the 2D parametric domain. If the trim curves are described by splines, they can be evaluated and converted to piecewise linear segments. In our implementation, the trim curves are evaluated and rendered directly to a trim-texture.

7.1. Trim-texture generation

As described by Woo et al. [21], 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. This is shown in Fig. 12, where only parts of the domain that are rendered once or thrice are considered to be the part of the surface that is to be finally rendered. Another advantage of using such an algorithm is that the orientation of the holes and holes within holes need not be explicitly considered.



(a) Using alpha blending. (b) Using fragment program.

Fig. 13. Difference in trimming using alpha blending versus a fragment program. Alpha blending produces incorrect results.

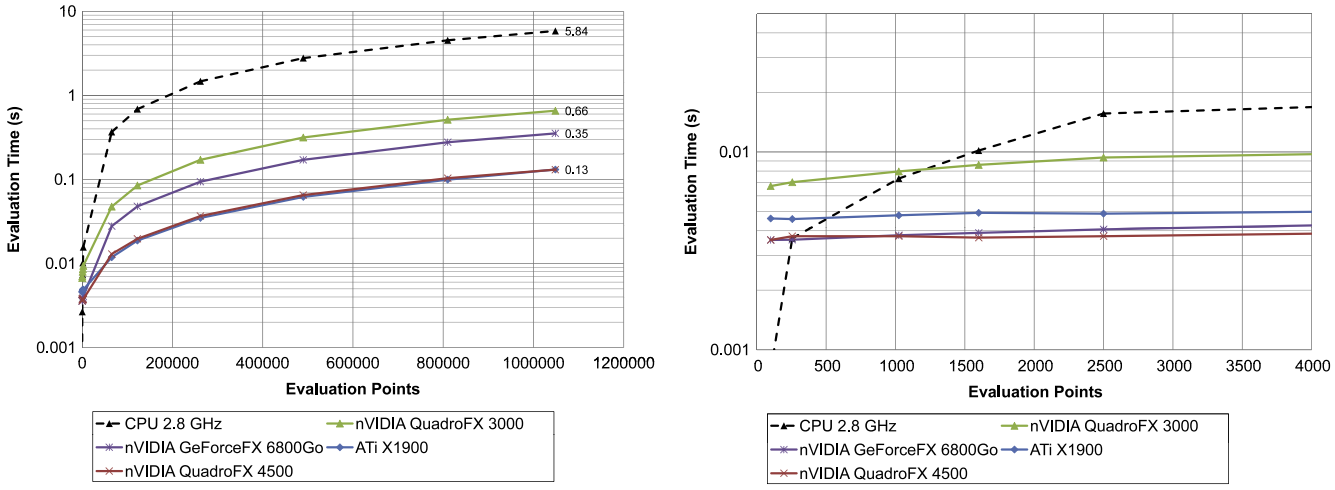
The above algorithm can be implemented either by using the stencil buffer or by alpha blending. Using the stencil buffer is sufficient to trim surfaces that are parallel to the view plane; implementation details for using the stencil buffer are given in [21]. However, we use an alternate implementation based on the alpha blending functionality of graphics cards to generate the trim-texture because the trimmed surfaces may be arbitrarily oriented or curved.

Some basic preprocessing is required for using alpha blending, as explained below. The viewport is set up to match the size of the trim-texture, which is determined based on the required LOD, as in [1]. The Model View matrix is set to 2D mode with view area from [0 1] in both width and height. For planar faces, the two directions correspond to the two orthogonal directions defining the coordinate system in the plane of the face; for non-planar faces, the parametric u and v directions that define the texture coordinate system are used. The background color is cleared to (0, 0, 0, 0). The required blending factors are chosen to perform an odd/even count. This can be done by toggling the existing value from 0 to 1 or 1 to 0 whenever a new fragment is drawn over it. Once all the parameters are set up, a triangle fan is drawn with color (1, 1, 1, 1). Thus, the algorithm can be easily extended to complex shapes such as fonts or irregular holes.

7.2. Rendering

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. One such example is shown in Fig. 13(a), where alpha blending is used to cut the holes for a scene with an airplane inside a box. The correct rendering is shown in Fig. 13(b). 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 may 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 (Fig. 13(b)). The advantage of



(a) Evaluation timings.

(b) Timings for small evaluation grids.

Fig. 14. Log-scale comparison of evaluation timings for a bi-cubic NURBS surface with increasing evaluation points.

Table 1
Different GPU platforms tested.

GPU	VRAM (MB)	CPU (GHz)	RAM (MB)
ATi X1900	512	2.8	512
nVIDIA Quadro FX4500	512	3.00	2048
nVIDIA Quadro FX3000	256	1.88	1024
nVIDIA GeForce FX6800Go	256	1.60	512

such a method is that the lighting calculations need not be done to those fragments that are discarded. However, this implementation uses branching and may lead to a performance drop in older graphic cards. Our fragment program used for the trimming operation, written in Cg [22,23], makes use of the *discard* command that kills the fragment when the value of the particular color channel used to trim is 0. To save memory we store different trim-textures in different color channels of the same texture. We then switch between the different channels while rendering different trimmed surfaces.

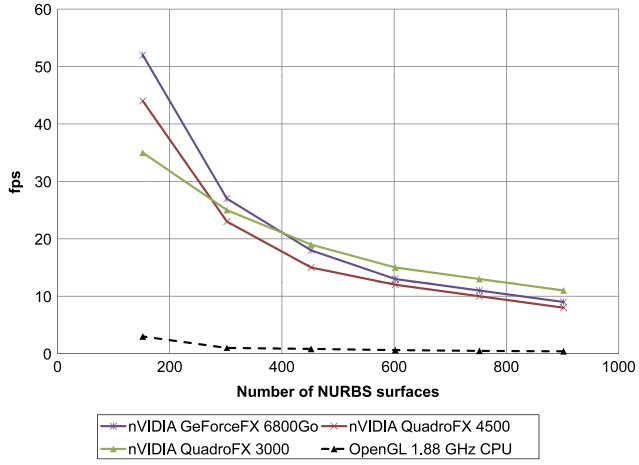


Fig. 15. Comparison of frame rates with different nVIDIA graphics cards. One-third of the total NURBS surfaces are non-trivially trimmed.

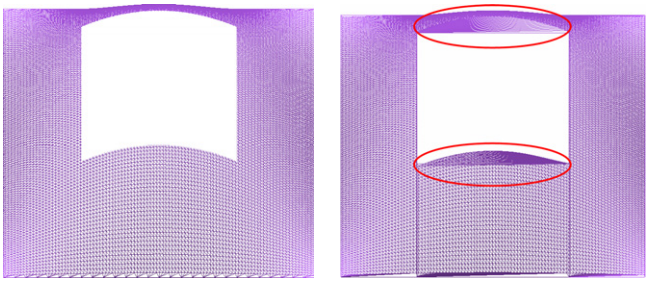
8. Results

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

Fig. 14(a) 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 evaluation time includes the time taken to generate the knot array and control point array on the CPU; the timings will remain the same even if the user interactively changes the knot values or the control points. The GPU-based evaluation is faster than the CPU-based evaluation by a factor of about 50 when evaluated at a large number 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 16×16 evaluation points (Fig. 14(b)). 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 Fig. 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. Fig. 1 is rendered using an evaluation grid of 64×64 points for each surface on a window of size 1280×1024. 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. This evaluation grid is similar to the one shown for the largest duck in Fig. 11. In addition, the model can be interactively displayed with varying LODs without re-sending the data to the GPU repeatedly. Similarly, any changes to the model will necessitate transferring only the control points to the GPU.

Fig. 15 compares the frame rates for an animated scene containing many such ducks swimming in a (tessellated) teapot, similar to Fig. 1, using our GPU implementation and with the CPU OpenGL implementation. The scene is again rendered in a window of size 1280×1024; the individual NURBS surfaces, being smaller than the full screen area, were evaluated on a 16×16 grid of evaluation points. One-third of the NURBS surfaces were non-trivially trimmed. 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. In addition, the OpenGL



(a) Correct surface. (b) OpenGL rendering.

Fig. 16. Trimmed NURBS surface rendered incorrectly by OpenGL. The figure on the left shows the correct trimming.

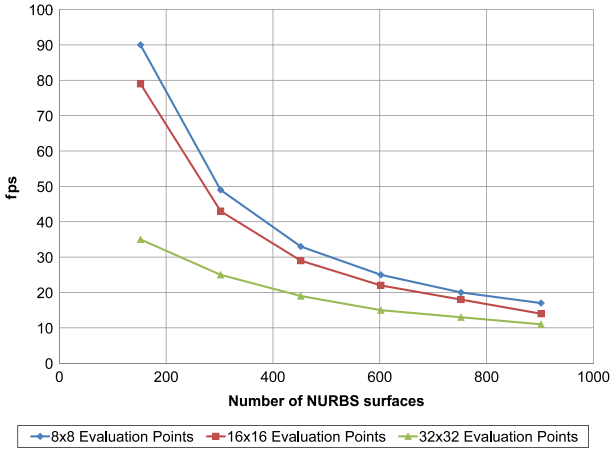


Fig. 17. Comparison of frame rates with varying per-patch evaluation grid size on nVIDIA Quadro FX3000 graphics card.

implementation had rendering artifacts at trim curve concavities while rendering trimmed NURBS surfaces (Fig. 16).

Fig. 17 shows the frame rates for animating the same scene as the above example but varying the per-patch evaluation grid size as well as the number of ducks. The frame rates were timed on the nVIDIA Quadro FX3000 graphics card. The NURBS surfaces evaluated on a 32×32 grid of evaluation points was the slowest, but for a larger number of surfaces the rates start to converge.

9. Summary and conclusions

We have presented a new method to evaluate and display trimmed NURBS surfaces on the GPU. Our algorithm evaluates the NURBS surface point coordinates directly, without resorting to approximations, using a unified evaluation framework that uses the same fragment program to evaluate arbitrary degree NURBS surfaces. Our evaluation framework that calculates all the basis function values in parallel can be extended to calculate derivatives and normals, serving as a foundation for modeling operations as well [24]. We show that packing the basis function arrays into the four color channels (along their height dimension to preserve the unified, degree-independent property of the implementation) yields a more efficient algorithm than unpacked or index-array based approaches. The method shows great promise for real-time interaction with exact NURBS models, as seen from the frame rates we achieved even on older graphics cards. The evaluation timings show more than 40 times improvement over evaluation on the CPU for large inputs, and a similar improvement in overall frame rate compared to the OpenGL implementation. However, this method is still not optimal for a small number of evaluation points since the overhead of setting up the GPU for performing the computations is high in this case. The number of surfaces that can be evaluated and displayed is primarily limited by texture memory on the GPU that is used to store the evaluated surface points and the trim data. We found our method to be capable of interactively evaluating and rendering up to 300 NURBS surfaces. 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

[1] Guthe M, Balázs A, Klein R. GPU-based trimming and tessellation of NURBS and T-spline surfaces. ACM Transactions on Graphics 2005;24(3):1016–23.
[2] Guthe M, Balázs A, Klein R. GPU-based appearance preserving trimmed NURBS rendering. Journal of WSCG 2006;14.
[3] Krishnamurthy A, Khardekar R, McMains S. Direct evaluation of NURBS curves and surfaces on the GPU. In: ACM symposium on solid and physical modeling. ACM; 2007. p. 329–34.
[4] Kanai T. Fragment-based evaluation of Non-Uniform B-spline surfaces on GPUs. Computer-Aided Design and Applications 2007;4(3):287–94.
[5] Kilgariff E, Fernando R. The GeForce 6 series GPU architecture. In: GPU gems 2 : Programming techniques for high-performance graphics and general-purpose computation. Addison-Wesley; 2005. 471–491.
[6] Owens JD, Luebke D, Govindaraju N, Harris M, Krüger J, Lefohn AE, et al. A survey of general-purpose computation on graphics hardware. Computer Graphics Forum 2007;26(1):80–113.
[7] Toth DL. On ray tracing parametric surfaces. ACM SIGGRAPH, vol. 85. 1985. p. 171–9.
[8] Nishita T, Sederberg TW, Kakimoto M. Ray tracing trimmed rational surface patches. ACM SIGGRAPH, vol. 90. 1990. p. 337–45.
[9] Martin W, Cohen E, Fish R, Shirley P. Practical ray tracing of trimmed NURBS surfaces. Journal of Graphics Tools 2000;5(1):27–52.
[10] Pabst H, Springer J, Schollmeyer A, Lenhardt R, Lessig C, Froehlich B. Ray casting of trimmed NURBS surfaces on the GPU, in: Proceedings of the IEEE symposium on interactive ray tracing, 2006. p. 151–160.
[11] Rockwood A, Heaton K, Davis T. Real-time rendering of trimmed surfaces. ACM SIGGRAPH, vol. 89. 1989. p. 107–16.
[12] Kumar S, Manocha D. Efficient rendering of trimmed NURBS surfaces. Computer-Aided Design 1995;27(7):509–21.
[13] Kumar S, Manocha D, Lastra A. Interactive display of large NURBS models. IEEE Transactions on Visualization and Computer Graphics 1996;2(4):323–36.
[14] Kahlesz F, Balázs A, Klein R. Multiresolution rendering by sewing trimmed NURBS surfaces, In: SMA '02: ACM symposium on solid modeling and applications. 2002. p. 281–288.
[15] Bolz J, Schröder P. Rapid evaluation of Catmull-Clark subdivision surfaces, in: Web3D 2002. 2002, pp. 11–17.
[16] Shiue L-J, Jones I, Peters J. A real-time GPU subdivision kernel. ACM Transactions on Graphics 2005;24(3):1010–5.
[17] Sederberg TW, Zheng J, Bakenov A, Nasri A. T-Splines and T-NURCCs. ACM Transactions on Graphics 2003;22(3):477–84.
[18] Sederberg TW, Zheng J, Sewell D, Sabin M. Non-uniform recursive subdivision surfaces. In: Computer graphics proceedings, annual conference series. ACM SIGGRAPH, vol. 98. 1998. p. 387–94.
[19] Loop C, Blinn J. Real-time GPU rendering of piecewise algebraic surfaces. ACM Transactions on Graphics 2006;25(3):664–70.
[20] Piegl LA, Tiller W. The NURBS book. 2nd ed. Springer; 1997.
[21] Woo M, Neider J, Davis T, Shreiner D. Drawing filled concave polygons using the stencil buffer. In: OpenGL(R) programming guide, version 1.4. 4th ed. Addison-Wesley; 2004. p. 600–1.
[22] Mark WR, Glanville RS, Akeley K, Kilgard MJ. Cg: A system for programming graphics hardware in C-like language. ACM Transactions on Graphics 2003; 22(3):896–907.
[23] Fernando R, Kilgard MJ. The Cg tutorial: The definitive guide to programmable real-time graphics. Boston: Addison-Wesley; 2003.
[24] Krishnamurthy A, Khardekar R, McMains S, Haller K, Elber G. Performing efficient NURBS modeling operations on the GPU. IEEE Transactions on Visualization and Computer Graphics 2009;15(4):530–43.