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.

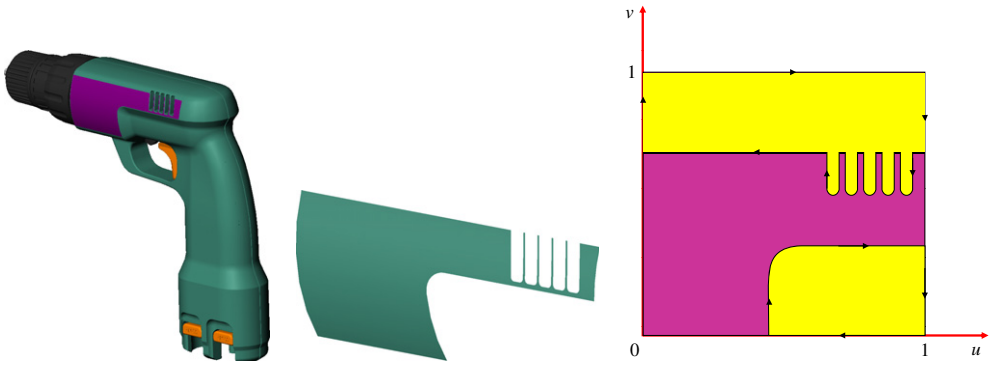
The fragment-program implementations of surface evaluation of subdivisions were not fast enough for real-time interaction with a large number of surfaces because the evaluated surface coordinates had to be read back from an off-screen pixel buffer using an expensive p-buffer switch for each surface. 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

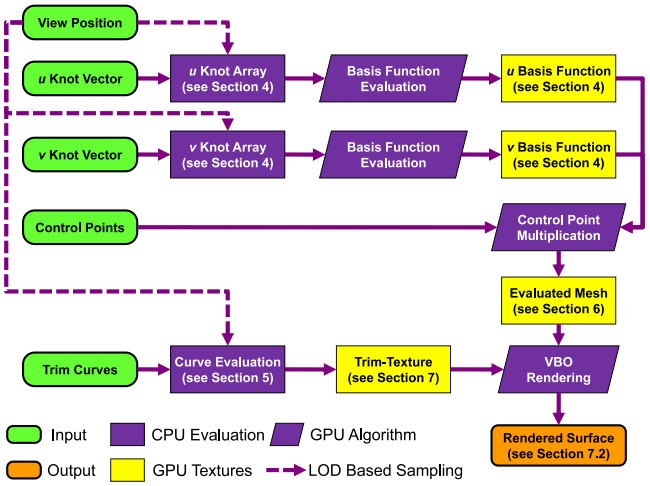


Fig. 3. Algorithm for rendering trimmed NURBS surface.

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].

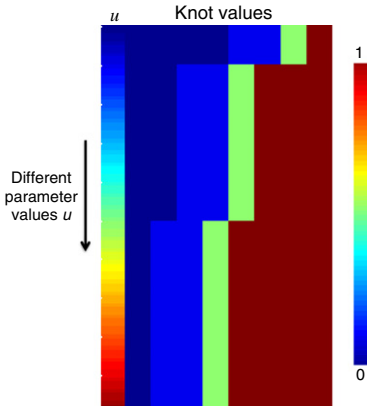


Fig. 4. Knot array; knot values to be transferred to the GPU as a texture visualized as a color plot. (For interpretation of the references to colour in this figure legend, the reader is referred to the web version of this article.)

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).

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

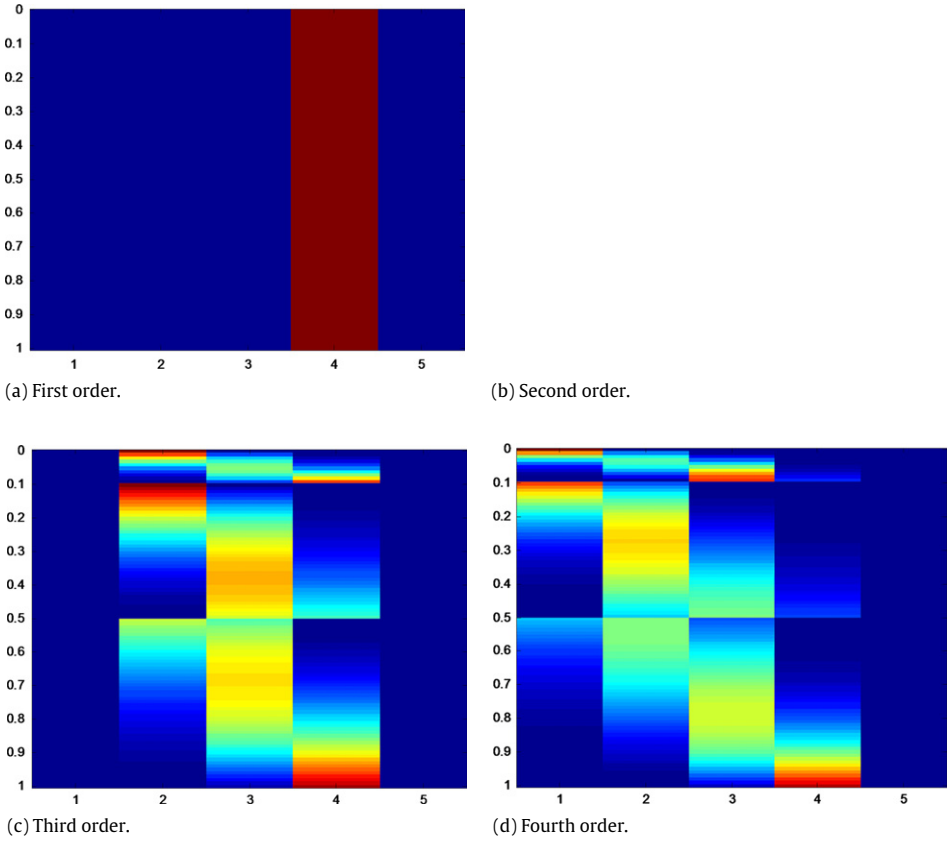
This vector is generated on the CPU; Fig. 5(a) shows the generated first-order basis functions for the cubic NURBS curve (order 4) for 100 evaluation points. The generated first-order basis function is then transferred to the graphics card and stored there as a texture, call it *tex*1.

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.)

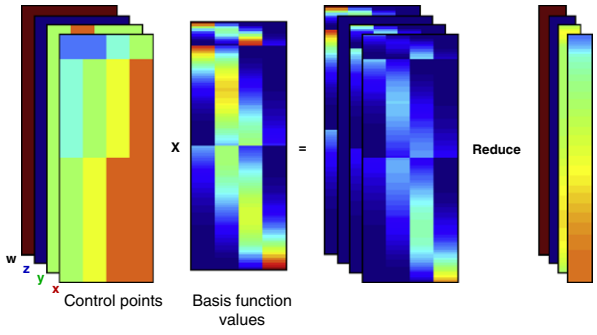


Fig. 6. Sequence of steps for curve point evaluation.

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