Appl. Math. J. Chinese Univ.
2014, 29(4): 455-467

Real-time accurate free-form deformation in terms of triangular Bézier surfaces

CUI Yuan-min FENG Jie-qing*

Abstract. We implemented accurate FFD in terms of triangular Bézier surfaces as matrix multiplications in CUDA and rendered them via OpenGL. Experimental results show that the proposed algorithm is more efficient than the previous GPU acceleration algorithm and tessellation shader algorithms.

§1 Introduction

Free-Form Deformation (abbreviated as FFD) is a widely used shape editing method in computer animation and geometric modeling [21]. It has been integrated in many commercial softwares due to its simpleness and intuitiveness. Without loss of generality, when deforming a mesh model with a B-spline volume, traditional FFD is applied to the points of the model that tends to produce aliased deformation result if sampling density is not high enough, whereas accurate FFD is applied to the faces of the model and the Bézier surfaces are obtained as an accurate deformation result [6, 7, 8].

Accurate FFD can generate high quality deformation results. But it contains intensive computations, such as B-spline volume evaluations, Bézier surface interpolations, Bézier surface tessellations, *etc.* Moreover, there is also huge data throughput between CPU and GPU or inside of GPU when displaying the tessellated Bézier surfaces. All of the above factors make real-time accurate FFD almost impossible for large scale models.

By fully exploiting parallel capacities of GPU, a new CUDA based GPU acceleration of accurate FFD algorithm is proposed in this paper. All of intensive computations are designed and implemented on GPU, and huge data transfer is also avoided. Experimental results show that the proposed algorithm is more efficient than the previous ones. The main contributions of the proposed algorithm include:

- Parallel evaluations of the B-spline volume on a regular grid.
- Reusable matrix multiplications for both interpolation and tessellation of triangular Bézier surfaces, which are fully implemented via CUBLAS on GPU.
- Adopting of vertex buffer objects for OpenGL rendering to reduce data transfer.

Received: 2013-10-21.
MR Subject Classification: 65D18, 68U05, 97R60.
Keywords: Accurate free-from deformation, GPU acceleration, CUDA, triangular Bézier surfaces.
Digital Object Identifier(DOI): 10.1007/s11766-014-3239-6.
Supported by the National Natural Science Foundation of China (61170138 and 61472349).

This paper is organized as follows: the related work is introduced in Section 2. The proposed GPU acceleration algorithm is described in detail in Section 3. The implementation results, comparisons and discussions are given in Section 4. Finally, conclusions are drawn and future research is indicated in Section 5.

§2 Related works

Free-Form Deformation is an intuitive method to manipulate and deform geometric models. It is widely used in computer animation and geometric modeling. FFD is first proposed by Sederberg and Parry [21]. The main idea of FFD is to embed an object into an intermediate volume, e.g. a Bézier, B-spline, or NURBS volume. Users edit the shape of the volume, then the deformation of the volume is transfered to the embedded object. Since the method is independent of geometric representation of the object, it is simple and intuitive. There are a lot of successive work on FFD. Most of them focus on the improvements of the original FFD from the aspects of intermediate volume shape, interactive means, etc. Gain and Bechmann [10] give a detailed survey of them. There are some new progress of FFD these years. For example, Xu, et al.[23] proposed a new method to manipulate the model directly based on curve-based FFD [14]. The main target of this algorithm is to minimize the change of the control polygons and the length of the model. In the mean time, the algorithm preserves geometry details of the model by using a method based on Laplacian coordinates.

Traditionally, FFD acts on the vertices of a model, thus the deformation result depends heavily on the sampling density of the deformed model. Many researchers try to solve this problem via an adaptive sampling way [20, 11, 9]. They take the surface curvature and polygon size into account and upsample the deformed model if necessary. Due to the intrinsic insufficiency of adaptive sampling methods, the proposed solutions cannot handle some special cases well.

Feng, et al. [6, 7, 8] proposed the concept of accurate FFD to solve the sampling problem in another way. Different from the traditional FFDs, accurate FFD deforms each face of the model as a triangular Bézier surface or trimmed Bézier surface patches based on Bernstein polynomials composition [3, 4]. At the costs of antialiased deformation results, accurate FFD contains intensive computations and huge data transfers. It is a challenge to achieve real-time accurate FFDs implemented on CPU.

With rapid developments of general-purpose GPU, some GPU based and efficient FFD methods are proposed in recent years. Chua, et al. [1] design a new GPU architecture dedicated to FFD. But none of vendors develops their GPU along this way. On the contrary, they are continuously enhancing the general purpose computing capabilities of GPU. Schein, et al. [22] implement a GPU accelerated FFD by NVIDIA CG language. Many programming tricks are adopted to overcome the GPU pipeline limitations which makes it not prevalent. Jung, et al. [15] achieve the same goal by NVIDIA CUDA, and embed it to the X3D system. Hahmann, et al. [12] propose a GPU-based volume-preserving FFD. They employ the multi-linear property of volume constraint and derive an explicit solution to the problem. Its GPU acceleration part which is implemented by CUDA is 6.5 times faster than its CPU counterpart. In the field of biomedicine, Modat, et al. [16] proposed a MRI non-rigid body registration method based on GPU accelerated FFD.

Recently, Cui, et al. [2] propose a real-time accurate FFD via CUDA, where the deformation result is represented in terms of trimmed tensor product Bézier patches. In fact, it is an optimal GPU implementation of method [8]. It can achieve high performance even for a large

scale model. However it tessellates the trimmed Bézier surfaces patch by patch, which is not an efficient way. In the CPU implementations, the approach of accurate FFD in terms of triangular Bézier surfaces [6, 7] is slower than the one in terms of trimmed Bézier surface patches [8]. However, after exploiting the parallel computing capacities of GPGPU and carefully analysing the accurate FFD approaches [6, 7], a more efficient GPU-based accurate FFD is proposed in this paper, where the object is accurately deformed as triangular Bézier surfaces [6, 7]. All the computations, tessellations and data transfer are implemented on GPU optimally. As a result, it can achieve higher performance than the method in [2].

§3 GPU Based accurate FFD in terms of triangular Bézier surfaces

3.1 Overview of accurate FFD algorithm in terms of triangular Bézier surfaces

Let $\mathbf{R}(u,v,w)$ be a B-spline volume of degree $n_u \times n_v \times n_w$. It has m_u, m_v and m_w control points along u, v and w directions respectively:

$$\mathbf{R}(u,v,w) = \sum_{i=0}^{m_u-1} \sum_{j=0}^{m_v-1} \sum_{k=0}^{m_w-1} \mathbf{R}_{ijk} N_{i,n_u}(u) N_{j,n_v}(v) N_{k,n_w}(w) \tag{1}$$

where $\{N_{i,n_u}(u)\}_{i=0}^{m_u-1}$, $\{N_{j,n_v}(v)\}_{j=0}^{m_v-1}$ and $\{N_{k,n_w}(w)\}_{k=0}^{m_w-1}$ are normalized B-spline basis functions. Its knot vectors along three directions are $\{u_i\}_{i=0}^{n_u+m_u}$, $\{v_j\}_{j=0}^{n_v+m_v}$ and $\{w_k\}_{k=0}^{n_k+m_k}$ respectively. Each three dimension region $[u_i, u_{i+1}] \times [v_j, v_{j+1}] \times [w_k, w_{k+1}]$ is called a knot box, where $n_u \le i < m_u$, $n_v \le j < m_v$ and $n_w \le k < m_w$ respectively.

As described in the papers [6, 7], firstly, the polygons in the object are subdivided against the knot boxes such that each of generated sub-polygons lies inside of a knot box. Secondly, non-triangular sub-polygons are triangulated. The accurate FFD of such a sub-triangle by $\mathbf{R}(u,v,w)$ is a triangular Bézier surface [6, 7], whose degree is $(n_u + n_v + n_w)$. Let the triangular Bézier surface be noted as $\mathbf{T}(u,v,w)$:

$$\mathbf{T}(u,v,w) = \sum_{\substack{i+j+k=n \\ 0 \le i,j,k \le n}} \mathbf{T}_{ijk} B^n_{ijk}(u,v,w), \quad u,v,w \ge 0, \quad u+v+w=1 \tag{2}$$

where $B^n_{i,j,k}(u,v,w) = \frac{n!}{i!j!k!} u^i v^j w^k$ is Bernstein basis function defined on a 2D simplex, i.e., triangle. The degree of $\mathbf{T}(u,v,w)$ is $n = n_u + n_v + n_w$ and its control points are written as $\{\mathbf{T}_{i,j,k} \mid i+j+k=n\}$.

3.2 Computing control points of triangular Bézier surface

$\mathbf{T}(u,v,w)$ has $m = (n+1)(n+2)/2$ control points, which can be expressed as a column vector $(\mathbf{T}_{n,0,0}\ \ \mathbf{T}_{n-1,1,0} \ldots \mathbf{T}_{0,0,n})^{\mathrm{T}}$.

According to [7], the control points can be computed efficiently via polynomial interpolation. To perform polynomial interpolation for $\mathbf{T}(u,v,w)$, $m = (n+1)(n+2)/2$ sampling points on $\mathbf{T}(u,v,w)$ should be known in advance. They can be obtained via B-spline volume $\mathbf{R}(u,v,w)$ evaluations, since the sampling points are on the deformed object in fact. For each triangular Bézier surface, uniform sampling points are: $\{\mathbf{T}(\frac{i}{n}, \frac{j}{n}, \frac{k}{n}) \mid 0 \le i,j,k \le n, i+j+k=n\}$, whose corresponding parameters for the B-spline volume are noted as $\{(\tilde{u}_i, \tilde{v}_i, \tilde{w}_i)\}_{i=1}^m$ Thus the m sampling points on $\mathbf{T}(u,v,w)$ are $\{\mathbf{R}(\tilde{u}_i, \tilde{v}_i, \tilde{w}_i)\}_{i=1}^m$ respectively. Next, the control points of $\mathbf{T}(u,v,w)$ can be calculated by interpolating these m sampling points. The interpolation can

be expressed as the following matrix multiplication:

$$\begin{pmatrix} \mathbf{T}_{n,0,0} \\ \mathbf{T}_{n-1,1,0} \\ \vdots \\ \mathbf{T}_{0,0,n} \end{pmatrix}_{m\times 1} = \mathbf{B}^{-1} \begin{pmatrix} \mathbf{R}(\tilde{u}_1, \tilde{v}_1, \tilde{w}_1) \\ \mathbf{R}(\tilde{u}_2, \tilde{v}_2, \tilde{w}_2) \\ \vdots \\ \mathbf{R}(\tilde{u}_m, \tilde{v}_m, \tilde{w}_m) \end{pmatrix}_{m\times 1} \tag{3}$$

where

$$\mathbf{B} = \begin{pmatrix} B^n_{n,0,0}(1,0,0) & \cdots & B^n_{0,0,n}(1,0,0) \\ B^n_{n,0,0}(\frac{n-1}{n}, \frac{1}{n}, 0) & \cdots & B^n_{0,0,n}(\frac{n-1}{n}, \frac{1}{n}, 0) \\ \vdots & \ddots & \vdots \\ B^n_{n,0,0}(0,0,1) & \cdots & B^n_{0,0,n}(0,0,1) \end{pmatrix}_{m\times m} \tag{4}$$

Because all the triangles of the model will be deformed as the triangular Bézier surfaces of degree n, their control points evaluation via formula (3) can be merged and rewritten as:

$$\begin{pmatrix} \mathbf{T}^1_{n,0,0} & \mathbf{T}^2_{n,0,0} & \cdots & \mathbf{T}^f_{n,0,0} \\ \mathbf{T}^1_{n-1,1,0} & \mathbf{T}^2_{n-1,1,0} & \cdots & \mathbf{T}^f_{n-1,1,0} \\ \vdots & \vdots & \ddots & \vdots \\ \mathbf{T}^1_{0,0,n} & \mathbf{T}^2_{0,0,n} & \cdots & \mathbf{T}^f_{0,0,n} \end{pmatrix}_{m\times f} = \mathbf{B}^{-1} \begin{pmatrix} \mathbf{R}_1(\tilde{u}^1_1, \tilde{v}^1_1, \tilde{w}^1_1) & \cdots & \mathbf{R}_f(\tilde{u}^f_1, \tilde{v}^f_1, \tilde{w}^f_1) \\ \vdots & \ddots & \ddots \\ \mathbf{R}_1(\tilde{u}^1_m, \tilde{v}^1_m, \tilde{w}^1_m) & \cdots & \mathbf{R}_f(\tilde{u}^f_m, \tilde{v}^f_m, \tilde{w}^f_m) \end{pmatrix}_{m\times f} \tag{5}$$

which can be further simplified and rewritten as:

$$\mathbb{T} = \mathbf{B}^{-1}\mathbb{R}_s \tag{6}$$

3.3 Uniform tessellations of triangular Bézier surfaces

After the triangular Bézier surfaces obtained, i.e., accurate FFD result, they will be tessellated for rendering. Here uniform tessellation is adopted for each triangular Bézier surface. It is a GPU friendly and efficient solution.

The uniform tessellation for a triangular Bézier surface is illustrated in Figure 1. The triangular Bézier surface is tessellated as m^2 triangles, where each edge of parametric domain is uniformly sampled $(m+1)$ points.

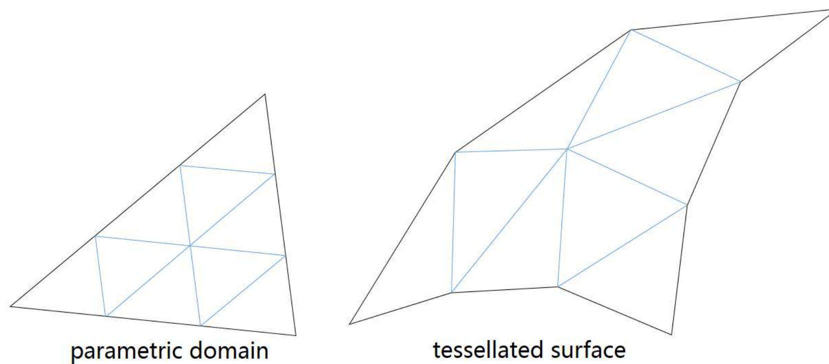


Figure 1: Uniform tessellation of a triangular Bézier surface

One of advantages of uniform tessellation is that the numbers and positions of tessellation

points on the adjacent edges are the same. Moreover, all the surfaces have the same tessellation pattern, so the matrix multiplications involved in the tessellation operation can be reused, which is in accordance with the SIMD architecture of GPU [17]. In the following, the detailed formulae for tessellation will be derived.

For the triangular Bézier surface $\mathbf{T}(u,v,w)$ in (2), a tessellation point corresponding to (u_0,v_0,w_0) can be evaluated:

$$\mathbf{T}(u_0,v_0,w_0) = \left[\begin{array}{ccc} \mathbf{B}^n_{n,0,0}(u_0,v_0,w_0) & \cdots & \mathbf{B}^n_{0,0,n}(u_0,v_0,w_0) \end{array}\right] \begin{pmatrix} \mathbf{T}_{n,0,0} \\ \mathbf{T}_{n-1,1,0} \\ \vdots \\ \mathbf{T}_{0,0,n} \end{pmatrix} \tag{7}$$

We assume that there are q points on the tessellated surface, their parameters are noted as $\{(u_i,v_i,w_i)\}_{i=1}^q$. The q tessellation points can be computed by the following formula:

$$\begin{pmatrix} \mathbf{T}(u_1,v_1,w_1) \\ \mathbf{T}(u_2,v_2,w_2) \\ \vdots \\ \mathbf{T}(u_q,v_q,w_q) \end{pmatrix}_{q\times 1} = \mathbf{B}_q \begin{pmatrix} \mathbf{T}_{n,0,0} \\ \mathbf{T}_{n-1,1,0} \\ \vdots \\ \mathbf{T}_{0,0,n} \end{pmatrix}_{m\times 1} \tag{8}$$

where

$$\mathbf{B}_q = \begin{pmatrix} B^n_{n,0,0}(u_1,v_1,w_1) & \cdots & B^n_{0,0,n}(u_1,v_1,w_1) \\ B^n_{n,0,0}(u_2,v_2,w_2) & \cdots & B^n_{0,0,n}(u_2,v_2,w_2) \\ \vdots & \ddots & \vdots \\ B^n_{n,0,0}(u_q,v_q,w_q) & \cdots & B^n_{0,0,n}(u_q,v_q,w_q) \end{pmatrix}_{q\times m} \tag{9}$$

Because all the triangular Bézier surfaces have the same tessellation pattern, formula (8) can be extended further for all f triangular Bézier surfaces and they can be merged as one matrix multiplication as follows:

$$\begin{aligned} &\begin{pmatrix} \mathbf{T}_1(u_1,v_1,w_1) & \mathbf{T}_2(u_1,v_1,w_1) & \cdots & \mathbf{T}_f(u_1,v_1,w_1) \\ \mathbf{T}_1(u_2,v_2,w_2) & \mathbf{T}_2(u_2,v_2,w_2) & \cdots & \mathbf{T}_f(u_2,v_2,w_2) \\ \vdots & \vdots & \ddots & \vdots \\ \mathbf{T}_1(u_q,v_q,w_q) & \mathbf{T}_2(u_q,v_q,w_q) & \cdots & \mathbf{T}_f(u_q,v_q,w_q) \end{pmatrix}_{q\times f} \\ =&\mathbf{B}_q \begin{pmatrix} \mathbf{T}^1_{n,0,0} & \mathbf{T}^2_{n,0,0} & \cdots & \mathbf{T}^f_{n,0,0} \\ \mathbf{T}^1_{n-1,1,0} & \mathbf{T}^2_{n-1,1,0} & \cdots & \mathbf{T}^f_{n-1,1,0} \\ \vdots & \vdots & \ddots & \vdots \\ \mathbf{T}^1_{0,0,n} & \mathbf{T}^2_{0,0,n} & \cdots & \mathbf{T}^f_{0,0,n} \end{pmatrix}_{m\times f} \end{aligned} \tag{10}$$

The above matrix multiplication can be written concisely as:

$$\mathbb{P} = \mathbf{B}_q \mathbb{T} \tag{11}$$

where $\mathbb{P}$ and $\mathbb{T}$ are tessellation points and control points of all triangular Bézier surfaces respectively.

To render the tessellated surfaces, the normals at the tessellation points should be evaluated. It is straightforward to evaluate tangent vectors at the tessellated points with respect to both u and v directions, and then compute their cross products as the normals. Before rendering, the normals should be normalized. Two tangent vectors can be evaluated via the following formulae:

$$\frac{\partial \mathbb{P}}{\partial u} = \frac{\partial \mathbf{B}_q}{\partial u}\mathbb{T}, \qquad \frac{\partial \mathbb{P}}{\partial v} = \frac{\partial \mathbf{B}_q}{\partial v}\mathbb{T} \tag{12}$$

They can be written concisely as:

$$\mathbb{P}_u = \mathbf{B}_{qu}\mathbb{T}, \quad \mathbb{P}_v = \mathbf{B}_{qv}\mathbb{T} \tag{13}$$

where $\mathbb{P}_u$ and $\mathbb{P}_v$ are the tangent vectors with respect to u and v respectively at the tessellation points.

3.4 GPU parallel computing sampling points $\mathbb{R}_s$ in B-spline volume

$\mathbb{R}_s$ in formula (6) has mf elements, each of them is a sampling point on the triangular Bézier surface, evaluated via the B-spline volume. There are two typical methods to evaluate a point in a B-spline volume [5]: de Boor-Cox algorithm and matrix multiplication algorithm. The former is numerically stable, but it is not suitable for GPU implementation since there are many nested loops in it, and it is also time-consuming. Thus the latter is preferable. Since all the mf elements in $\mathbb{R}_s$ are independent of each other, each thread can calculate one of the elements. It is in accordance with the SIMD structure of GPU, i.e., Single Instruction and Multiple Data [17].

Because all the triangular Bézier surfaces correspond to the same number of sampling points, i.e., the number of control points of the triangular surface, the task allocation is straightforward: assume the thread number of each thread block is *blockDim*, then the number of thread blocks is $\lceil mf/blockDim \rceil$. The thread with index $(blockIdx, threadIdx)$ can be labeled with a global index:

$$globalIdx = blockDim * blockIdx + threadIdx \tag{14}$$

According to (14), the sampling point with index *globalIdx* belongs to the $\lfloor globalIdx/m \rfloor th$ triangular Bézier surface, and it is the $(globalIdx\%m)th$ sampling point in it. If $globalIdx \geq mf$, the thread will be idle. The above strategy guarantees each CUDA thread evaluates one B-spline point. The workload is balanced, and the number of the idle threads is less than *blockDim*. In this way, GPU parallel computing capacity is fully exploited.

3.5 Efficient tessellation points evaluation algorithm

After the sampling points obtained, the triangular Bézier surfaces, i.e., accurate FFD result, can be obtained via polynomial interpolation according to formula (6). To display the surfaces, the tessellation points on the surface should be evaluated according to formula (11). By substituting (6) into (11), we can obtain:

$$\mathbb{P} = \mathbf{B}_q\mathbf{B}^{-1}\mathbb{R}_s \tag{15}$$

where $\mathbb{R}_s$ denotes all sampling points, $\mathbf{B}^{-1}$ is the matrix for triangular surface interpolation, $\mathbf{B}_q$ is the matrix of Bernstein polynomials for triangular Bézier surface evaluation. It is similar to the tangent vector evaluations, which are described in formula (13):

$$\mathbb{P}_u = \mathbf{B}_{qu}\mathbf{B}^{-1}\mathbb{R}_s, \quad \mathbb{P}_v = \mathbf{B}_{qv}\mathbf{B}^{-1}\mathbb{R}_s \tag{16}$$

There are two methods to evaluate (15) and (16). One method is to evaluate the control points of the triangular Bézier surfaces $\mathbb{T} = \mathbf{B}^{-1}\mathbb{R}_s$ first, and then evaluate the tessellation points and their tangent vectors:

$$\mathbb{P} = \mathbf{B}_q\mathbb{T}, \quad \mathbb{P}_u = \mathbf{B}_{qu}\mathbb{T}, \quad \mathbb{P}_v = \mathbf{B}_{qv}\mathbb{T} \tag{17}$$

The other one is to evaluate the product of first two matrices and obtain three intermediate matrix $\mathbf{B}_b$, $\mathbf{B}_{bu}$ and $\mathbf{B}_{bv}$ as follows:

$$\mathbf{B}_b = \mathbf{B}_q\mathbf{B}^{-1}, \quad \mathbf{B}_{bu} = \mathbf{B}_{qu}\mathbf{B}^{-1}, \quad \mathbf{B}_{bv} = \mathbf{B}_{qv}\mathbf{B}^{-1} \tag{18}$$

and then evaluate the tessellation points and their tangent vectors via $\mathbf{B}_b$, $\mathbf{B}_{bu}$ and $\mathbf{B}_{bv}$ as follows:

$$\mathbb{P} = \mathbf{B}_b\mathbb{R}_s, \quad \mathbb{P}_u = \mathbf{B}_{bu}\mathbb{R}_s, \quad \mathbb{P}_v = \mathbf{B}_{bv}\mathbb{R}_s \tag{19}$$

Intuitively, the complexities of the two methods seem the same. After carefully analysis, it shows that they are different and suitable for different cases.

Method 1: The triangular Bézier surface, $\mathbb{T} = \mathbf{B}^{-1}\mathbb{R}_s$ is evaluated first. $\mathbb{T}$ has mf elements, where m is the number of control points of the triangular Bézier surface, f is the number of the triangular Bézier surfaces. Each of them can be obtained via m MADs (multiply and add operation). Thus the overall MAD number in $\mathbb{T} = \mathbf{B}^{-1}\mathbb{R}_s$ is $m^2 f$. After $\mathbb{T}$ is obtained, $\mathbb{P}$, $\mathbb{P}_u$ and $\mathbb{P}_v$ are evaluated via (17).

Among them, $\mathbb{P}$ has qf elements, that q is the number of tessellation points of each surface, evaluating an element requires m MADs. Thus the overall MAD number for all tessellation points is mqf. To exploit the parallel computing capacity, the matrices $\mathbf{B}_{qu}$ and $\mathbf{B}_{qv}$ are expanded as the $q \times m$ ones with zero column vector, such that $\mathbb{P}_u$ and $\mathbb{P}_v$ have the same number of MADs with $\mathbb{P}$. In summary, **Method 1** requires ($mad_1 = m^2 f + 3mqf$) MADs. It is noted that the evaluation of B-spline volume point for each element in $\mathbb{R}_s$ is omitted, whose complexity is a constant for two methods.

Method 2: Three matrices $\mathbf{B}_b$, $\mathbf{B}_{bu}$ and $\mathbf{B}_{bv}$ in (18) are evaluated first. $\mathbf{B}_b$ has mq elements. Evaluating one element of $\mathbf{B}_b$ requires m MADs. Thus the overall MAD number for evaluating $\mathbf{B}_b$ is $m^2 q$. $\mathbf{B}_{bu}$ and $\mathbf{B}_{bv}$ have the same number of MADs with $\mathbf{B}_b$ due to the same above mentioned reason.

Similar to **Method 1**, evaluating the tessellation points and their tangent vectors $\mathbb{P}$, $\mathbb{P}_u$ and $\mathbb{P}_v$ require $3mqf$ MADs overall. In summary, Method 2 requires $mad_2 = (3m^2 q + 3mqf)$ MADs.

To compare two methods, we obtained

$$mad_1 - mad_2 = m^2 f - 3m^2 q = m^2(f - 3q) \tag{20}$$

where f is the number of triangular Bézier surfaces in accurate FFD result, q is the number of tessellation points on each triangular Bézier surface. If $f > 3q$, i.e., the number of surfaces is greater than three times of the number of tessellation points, then **Method 2** is more efficient than **Method 1**. Otherwise, $f < 3q$, i.e., a coarse mesh model that contains dozens of vertices, then **Method 1** is better. In practical application, $f > 3q$ is the common case. Thus **Method 2** is preferable, i.e., six matrix multiplications in (18) and (19). They can be further optimized respectively as follows.

3.6 GPU implementation of triangular Bézier surfaces tessellations

In general, the degree of B-spline is $1 \leq n_u, n_v, n_w \leq 3$ which can satisfy most of requirements in practical applications. Thus, the degree of the corresponding triangular Bézier surface is $3 \leq n \leq 9$. As described before, the matrix $\mathbf{B}^{-1}$ in (4) for surface interpolation only depends on the surface's degree, thus $\{\mathbf{B}_i^{-1}\}_{i=3}^{9}$ can be precomputed and stored. The matrices $\mathbf{B}_q$, $\mathbf{B}_{qu}$ and $\mathbf{B}_{qv}$ in (11) and (13) for tessellation only depend on the degree of the triangular Bézier surface and the tessellation pattern. They can also be precomputed and stored if the B-spline volume's degree and tessellation pattern are determined. The three matrices $\mathbf{B}_q$, $\mathbf{B}_{qu}$ and $\mathbf{B}_{qv}$ remain unchanged when the shape of the B-spline volume is edited.

The CUBLAS is a library of optimized implementation of BLAS on GPU [18]. BLAS is an API for basic linear algebra operations, *e.g.* vector and matrix multiplications. It is 6∼17 times faster than its CPU counterpart. Thus the CUBLAS is adopted to perform the above matrix

multiplications. The CUBLAS supports four kinds of data types: float, double, float complex, double complex. The matrix multiplications in (18) can be implemented straightforwardly by calling CUBLAS. In (19), the sampling points and their tangent vectors $\mathbb{P}$, $\mathbb{P}_u$ and $\mathbb{P}_v$ are in terms of 3D point, thus the matrix multiplications in (19) should be decomposed into 9 componentwise matrix multiplications corresponding to x, y, z components respectively:

$$\mathbb{P}_x = \mathbf{B}_b\mathbb{R}_{sx}, \quad \mathbb{P}_y = \mathbf{B}_b\mathbb{R}_{sy}, \quad \mathbb{P}_z = \mathbf{B}_b\mathbb{R}_{sz} \tag{21}$$

$$\mathbb{P}_{ux} = \mathbf{B}_{bu}\mathbb{R}_{sx}, \quad \mathbb{P}_{uy} = \mathbf{B}_{bu}\mathbb{R}_{sy}, \quad \mathbb{P}_{uz} = \mathbf{B}_{bu}\mathbb{R}_{sz} \tag{22}$$

$$\mathbb{P}_{vx} = \mathbf{B}_{bv}\mathbb{R}_{sx}, \quad \mathbb{P}_{vy} = \mathbf{B}_{bv}\mathbb{R}_{sy}, \quad \mathbb{P}_{vz} = \mathbf{B}_{bv}\mathbb{R}_{sz} \tag{23}$$

After having finished all these matrix multiplications by using function cublasSgemm provided by CUBLAS, the results, i.e., tessellation points and their normals, are stored into a Vertex Buffer Object(VBO), and then are rendered by the OpenGL efficiently. Because the formats of the matrix multiplication results of (21), (22) and (23) are not compatible with the VBO, a copy function is called to rearrange them. The copy operation is straightforward: each CUDA thread takes charge in one tessellation point, first calculates the cross product of the tangent vectors with respect to u and v directions then normalize the cross product as the normal, finally copy the coordinates and normals of the tessellation point to the VBO.

3.7 The optimization of GPU tessellation

In Section 3.6, the CUBLAS is adopted to accomplish matrix multiplications in the triangular Bézier surface tessellations. In SIMD architecture of GPU, combining several small matrix multiplications to a large one can obtain a better performance, since it is a more intensive data operation. According to this fact, the 12 matrix multiplications in (18), (21), (22) and (23) can be further optimized as two ones:

$$\begin{pmatrix} \mathbf{B}_b \\ \mathbf{B}_{bu} \\ \mathbf{B}_{bv} \end{pmatrix}_{3q\times m} = \begin{pmatrix} \mathbf{B}_q \\ \mathbf{B}_{qu} \\ \mathbf{B}_{qv} \end{pmatrix}_{3q\times m} \begin{pmatrix} \mathbf{B}^{-1} \end{pmatrix}_{m\times m} \tag{24}$$

$$\begin{pmatrix} \mathbb{P}_x & \mathbb{P}_y & \mathbb{P}_z \\ \mathbb{P}_{ux} & \mathbb{P}_{uy} & \mathbb{P}_{uz} \\ \mathbb{P}_{vx} & \mathbb{P}_{vy} & \mathbb{P}_{vz} \end{pmatrix}_{3q\times 3f} = \begin{pmatrix} \mathbf{B}_b \\ \mathbf{B}_{bu} \\ \mathbf{B}_{bv} \end{pmatrix}_{3q\times m} \begin{pmatrix} \mathbb{R}_{sx} & \mathbb{R}_{sy} & \mathbb{R}_{sz} \end{pmatrix}_{m\times 3f} \tag{25}$$

§4 Implementation results

The proposed algorithm is implemented on a PC with Intel Core i5 760@2.8GHz CPU, 4GB Memory, NVIDIA GeForce GTX 465 GPU. The operating system is Arch Linux x86_64. The CPU part of our program is written by C++ and the GPU part is written by CUDA C. Moreover, the direct manipulation algorithm proposed by Hu *et al.* [13] is integrated into our system. In Figures 2-6, all the results are deformed by B-spline volumes of degree $2 \times 2 \times 2$. Each triangular Bézier surface is tessellated into 100 subtriangles. Among which the model in Figure 2 is manipulated directly via Hu's method.

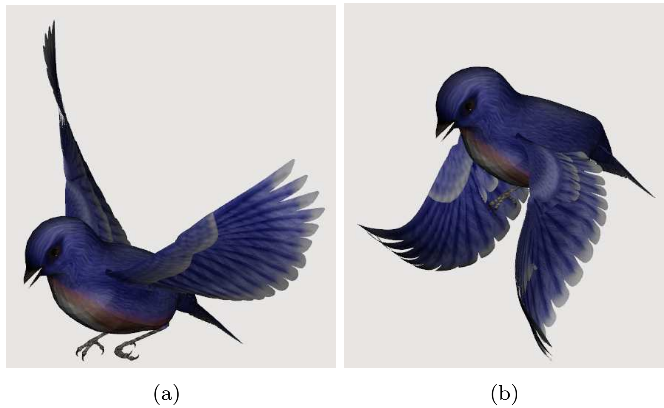
4.1 The optimized matrix multiplications

Table 1 gives the statistics of direct CUBLAS implementations of (18), (21∼23) and optimized implementations (24) and (25) for examples. It shows that the latter is 25%∼30% faster

Table 1: The efficiency comparison between two matrix multiplication methods in section 3.6 and section 3.7 (ms)

model	face number	algorithm	formula (24)	formula (25)	total	(trivial - optimal) / trivial
bird (Fig.2)	17119	trivial	0.024	3.954	3.978	28.66%
		optimal	0.011	2.827	2.838	
amphora (Fig.3)	19924	trivial	0.024	4.898	4.922	28.08%
		optimal	0.011	3.529	3.540	
snail (Fig.4)	46742	trivial	0.024	9.861	9.885	26.84%
		optimal	0.011	7.221	7.232	
spaceship (Fig.5)	96314	trivial	0.024	20.225	20.249	26.25%
		optimal	0.012	14.921	14.933	
tree (Fig.6)	113663	trivial	0.024	22.707	22.731	25.87%
		optimal	0.011	16.839	16.850	

than the former one. The reason is that (24) and (25) are more data intensive operations than (18), (21∼23), which is more suitable for GPU.



(a) (b)

Figure 2: A bird and its deformation

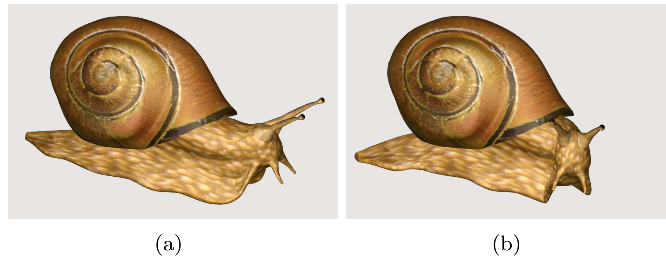
4.2 Comparison of the tessellation shader algorithm to CPU algorithm

The proposed algorithm is implemented by CPU too. Compared with the CPU implementation of the accurate FFD in terms of triangular Bézier surfaces [7], the CPU implementation of the proposed algorithm is also optimized for matrix multiplications as indicated in Section 3.7. Besides the sampling points in the B-spline volume evaluated by CPU, the rest of intensive computations involved in the evaluation and tessellation of triangular Bézier surfaces are abstracted as matrix multiplications (24) and (25) and implemented by CBLAS library on CPU. The results are given in Table 2.

Tessellation shader is a new GPU pipeline feature which can efficiently tessellate a patch into many micropatches. Thus, it's also suitable for rendering the triangular Bézier surfaces.

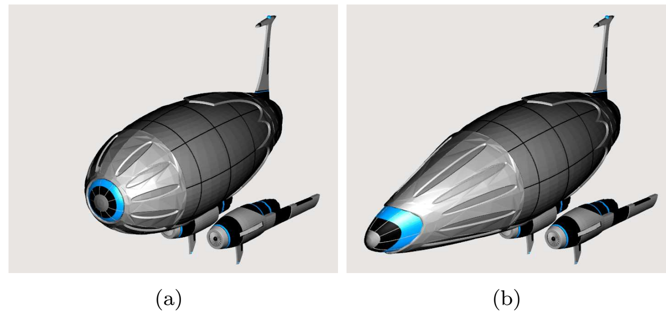
(a) (b)

Figure 3: An amphora and its deformation



(a) (b)

Figure 4: A snail and its deformation

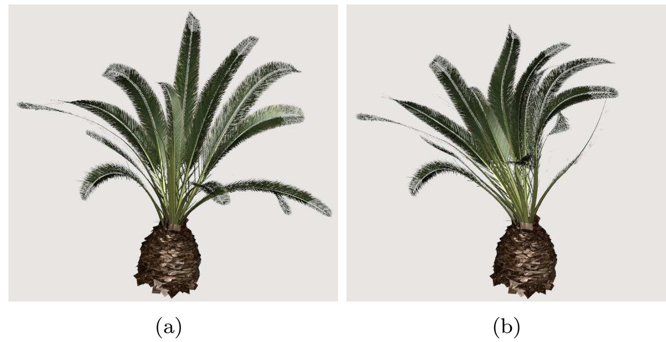


(a) (b)

Figure 5: A spaceship and its deformation

We also implemented our algorithm by using tessellation shaders. Firstly, the control points of all the triangular Bézier patches are evaluated as before. Secondly, the triangular surfaces are tessellated via tessellation shaders. The runtime comparison between the proposed CUDA algorithm and the tessellation shader algorithm is shown in Table 2.

From Table 2 we can see the proposed CUDA algorithm is 1.6∼1.7 times faster than the



(a) (b)

Figure 6: A tree and its deformation

Table 2: The efficiency comparison between the proposed CUDA algorithm, tessellation shader algorithm and CPU algorithm (ms)

model	CUDA(ms)	tessellation shader(ms)	shader/CUDA	CPU(ms)	CPU/CUDA
bird	12.757	33.657	2.638	661	51.81
amphora	15.468	41.568	2.687	831	53.72
snail	32.333	86.324	2.670	1707	52.79
spaceship	64.885	179.261	2.763	3528	54.37
tree	74.458	201.758	2.710	4003	53.76

tessellation shader algorithm and over 50 times faster than the CPU algorithm.

4.3 Comparison to the accurate FFD algorithm in terms of trimmed Bézier surface patches [2]

Table 3 gives the detailed statistics of the proposed algorithm and the GPU accurate FFD algorithm [2], where the result is in terms of trimmed tensor product Bézier surface patches. Essentially, the two accurate FFD results are identical expect for their different geometric representations. Since two algorithms are not identical, the steps in one algorithm are not one to one correspondence of the steps in the other algorithm. But from the overall runtime statistics, we can conclude that the proposed algorithm is 30%∼40% faster than the tensor product algorithm. If the runtime of control points evaluation and tessellation are considered only, the proposed algorithm is almost twice as fast as the tensor product algorithm.

The acceleration reason lies that the accurate FFD result is represented in terms of triangular Bézier surfaces of the same degree. The surfaces are tessellated with the same tessellation pattern. Compared to the tensor product algorithm [2], the operations in the proposed algorithm is more data-intensive, which is suitable for the SIMD architecture of the contemporary GPU.

§5 Conclusion

In this paper a more efficient GPU acceleration algorithm of accurate FFD is proposed, where the deformation result is composed of triangular Bézier surfaces. Through simple formula derivation, the computations involved in the accurate FFD are abstracted to two-stage

Table 3: The comparison of tensor product algorithm and the proposed algorithm (ms).

tensor product algorithm [2]						
model (triangles)	edit B-spline volume	copy data from CPU to GPU	calculate $\mathbf{S}_{n_s+1}^{-1}\mathbf{R}(\mathbf{T}_{n_t+1}^{\mathrm{T}})^{-1}$	tessellated trimmed Bézier patches	render patches	total
bird (17119)	< 1	0.009	4.138	9.652	6	19.799
amphora (19924)	< 1	0.009	4.171	11.067	7	22.247
snail (46742)	< 1	0.009	10.916	24.920	16	51.845
spaceship (96314)	< 1	0.009	20.675	49.871	32	102.555
tree (113663)	< 1	0.009	25.936	58.286	37	121.231

the proposed algorithm										
model (triangles)	edit B-spline volume	copy data from CPU to GPU	evaluate sampling points $\mathbb{R}_s$	tessellate surfaces	tessellate surfaces: formula (24)	tessellate surfaces: formula (25)	tessellate surfaces: copy result to VBO	render surfaces	total	total: acceleration ratio *
bird (17119)	< 1	0.009	2.601	3.961	0.011	2.840	1.110	6	12.571	36.51%
amphora (19924)	< 1	0.009	3.266	4.962	0.011	3.558	1.393	7	15.237	31.51%
snail (46742)	< 1	0.009	6.680	10.148	0.011	7.274	2.863	16	32.837	36.66%
spaceship (96314)	< 1	0.009	13.799	20.946	0.011	15.025	5.910	32	66.754	34.91%
tree (113663)	< 1	0.009	15.601	23.633	0.011	16.962	6.660	37	76.243	37.11%

* calculated by $(t_1 - t_2)/t_1$ in which t_1 is the total executing time of tensor product algorithm and t_2 is that of the proposed algorithm.

matrix multiplications. They can be accomplished by optimized GPU BLAS: CUBLAS. The experimental results show that the proposed algorithm is more efficient than the GPU-based accurate FFD algorithm in terms of trimmed tensor-product Bézier surface patches [2]. The main reason for obtain the performance improvement is that the proposed parallel algorithm is designed in a data intensive way instead of a instruction intensive way. The GPGPU architecture is more suitable for this kind of parallel tasks.

The proposed algorithm can be further improved from several aspects. First, adaptive tessellation can save graphics memory, since it can prevent the generation of too many tiny triangles or flat patches. However, it is complex to be implemented on GPU. Second, the algorithm is based on CUDA, so it can be only implemented on NVIDIA GPUs. If it is implemented by OpenCL [19], the universal property will be much better.

References

[1] C Chua, U Neumann. *Hardware-accelerated free-form deformation*, In: *ACM SIGGRAPH Workshop on Graph Hardware* (Interlaken, Seitzerland, 2000), HWWS Conf Proc, 2000, 33-39.

[2] Y Cui, J Feng. *Real-time B-spline Free-Form Deformation via GPU acceleration*, Comput Graph, 2013, 37(12): 1-11.

[3] T DeRose. *Composing Bézier simplexes*, ACM Trans Graph, 1988, 7(3): 198-221.

[4] T DeRose, R Goldman, H Hagen, S Mann. *Functional composition algorithms via blossoming*, ACM Trans Graph, 1993, 12(2): 113-135.

[5] G Farin. *Curves and surfaces for CAGD: a practical guide*, San Francisco, Morgan Kaufmann Publishers Inc., 2002.

[6] J Feng, P A Heng, T T Wong. *Accurate B-spline free-form deformation of polygonal objects*, J Graph Tools, 1998, 3(3): 11-27.

[7] J Feng, Q Peng. *Accelerating accurate b-spline free-form deformation of polygonal objects*, J Graph Tools, 2000, 5(1): 1-8.

[8] J Feng, T Nishita, X Jin, Q Peng. *B-spline free-form deformation of polygonal object as trimmed Bézier surfaces*, Visual Comput, 2002, 18: 493-510.

[9] J E Gain, N A Dodgson. *Adaptive Refinement and Decimation under Free-Form Deformation*, In: *Eurographics* (Cambridge, 1999), Eurograph Conf Proc, 1999, 17: 13-15.

[10] J E Gain, D Bechmann. *A survey of spatial deformation from a user-centered perspective*, ACM Trans Graph, 2008, 27(4): 1-21.

[11] J Griessmair, W Purgathofer. *Deformation of Solids with Trivariate B-Splines*, In: *Eurographics* (North Holland, 1989), Eurograph Conf Proc, 1989, 137-148.

[12] S Hahmann, G P Bonneau, S Barbier, G Elber, H Hagen. *Volume-preserving FFD for programmable graphics hardware*, Visual Comput, 2012, 28: 231-245.

[13] S M Hu, H Zhang, C L Tai, J G Sun. *Direct manipulation of FFD: efficient explicit solutions and decomposible multiple point constraints*, Visual Comput, 2001, 17(6): 370-379.

[14] K C Hui. *Free-form design using axial curve-pairs*, Comput Aided Design, 2002, 34(8): 583-595.

[15] Y Jung, H Graf, J Behr, A Kuijper. *Mesh Deformations in X3D via CUDA with Freeform Deformation Lattices*, Virtual and Mixed Reality - Systems and Applications, Lecture Notes in Comput Sci, 2011(6774): 343-351.

[16] M Modat, G R Ridgway, Z A Taylor, A Zeike, M Lehmann, J Barnes, D J Hawkes, N C Fox, S Ourselin. *Fast free-form deformation using graphics processing units*, Comput Methods Prog Biomed, 2010, 98(3): 278-284.

[17] NVIDIA. *Chapter 1: Introduction*, CUDA C PROGRAMMING GUIDE, 2014.

[18] NVIDIA. *CUBLAS*, Nvidia developer zone, https://developer.nvidia.com/cublas, 2013.

[19] KHRONOS. *OpenCL*, OpenCL overview, http://www.khronos.org/opencl/, 2013.

[20] S R Parry. *Free-form deformation in a constructive solid geometry modeling system*, Ph.D Thesis, Brigham Young University, 1986.

[21] T W Sederberg, S R Parry. *Free-form deformation of solid geometric models*, SIGGRAPH Comput Graph, 1986, 20(4): 151-160.

[22] S Schein, G Elber. *Real-time Free-form Deformation using Programmable Hardwares*, Int J Shape Model, 2006, 12: 179-192.

[23] G Xu, K C Hui, W B Ge, G Z Wang. *Direct manipulation of free-form deformation using curve-pairs*, Comput Aided Design, 2013, 45(3): 605-614.

State Key Lab of CAD&CG, Zhejiang University, Hangzhou 310058, China
Email: jqfeng@cad.zju.edu.cn