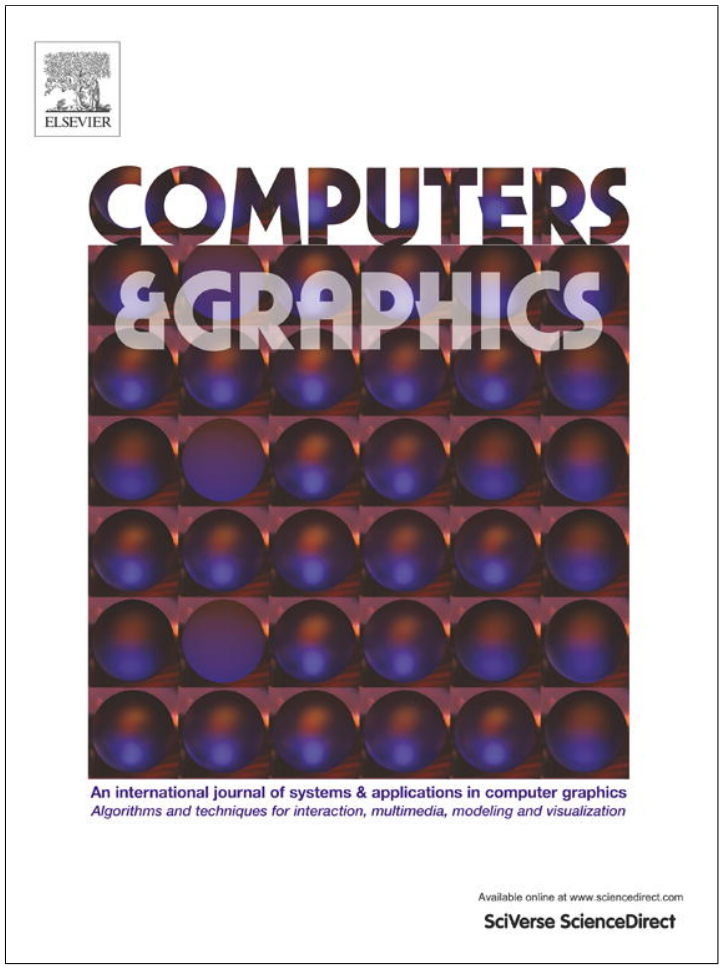


(This is a sample cover image for this issue. The actual cover is not yet available at this time.)

Computers & Graphics 37 (2013) 1–11

Contents lists available at SciVerse ScienceDirect

Computers & Graphics

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

Technical Section

# Real-time B-spline Free-Form Deformation *via* GPU acceleration

Yuanmin Cui, Jieqing Feng*

*State Key Lab of CAD&CG, Zhejiang University, 310058, China*

ARTICLE INFO

*Article history:*
Received 11 May 2012
Received in revised form
1 December 2012
Accepted 3 December 2012
Available online 12 December 2012

*Keywords:*
Accurate free-from deformation
GPU acceleration
CUDA

ABSTRACT

Accurate Free-Form Deformation is an analytical solution of deformation sampling, where a polygonal object is deformed as a set of trimmed Bézier surfaces. However, the operation is far from being interactive due to its high computational cost. In this paper, we proposed a real-time B-spline Free-Form Deformation of polygonal objects *via* GPU acceleration. Various time-consuming evaluations are designed and performed by means of highly parallel processing on GPGPU, such as evaluations of points of B-spline volume, calculations of control points of Bézier surfaces, tessellations of trimmed Bézier surfaces, evaluations of normals of tessellated triangles, *etc*. The adoption of vertex buffer object for rendering the tessellated trimmed Bézier surfaces greatly saves data I/O throughput. Experimental results show that the proposed GPU algorithm gains more than 200 times acceleration than its CPU counterpart.



## 1. Introduction

Free-Form Deformation (abbreviated as FFD) is widely applied in geometric modeling and computer animation. As an intuitive and versatile shape-editing tool, it has been integrated into mainstream commercial 3D computer animation software, such as Maya, Softimage XSI, 3DS Max, *etc.* As an improvement of FFD of polygonal objects, Accurate FFD deforms planar polygons as trimmed Bézier surface patches [1,2], whereas traditional FFD deforms points as points. Thus it can generate an accurate and natural deformation result, which is free of sampling problems in the deformed polygonal object. Furthermore, the representation of trimmed Bézier surface patches is consistent with the industrial standard STEP [3]. Thus the Accurate FFD can be easily integrated into current animation and geometric modeling systems.

However, the computational cost of Accurate FFD is very high. It includes a large number of trivariate B-spline volume evaluations and matrix multiplications. The tessellations and rendering of the trimmed Bézier surfaces are also heavy burdens on both CPU and GPU (Graphics Processing Unit). Thus the interactive Accurate FFD is almost prohibited.

In this paper, we propose a GPU-based interactive Accurate FFD algorithm *via* CUDA (Compute Unified Device Architecture). All of the computation-intensive and rendering tasks are designed and implemented on the GPU using the parallel computing API, CUDA. Finally we can achieve interactive Accurate FFD of polygonal objects. The main contributions of the paper are:

- GPU computations of control points of Bézier surfaces *via* more numerically stable sampling and reconstruction.
- Full GPU tessellations of trimmed Bézier surfaces.
- Efficient rendering of tessellated trimmed Bézier surfaces *via* a vertex buffer object.

The structure of this paper is as follows: Section 2 provides an overview of previous FFD methods. In Section 3, we discuss the details of our GPU-accelerated Accurate FFD algorithm. Section 4 presents our implementation results and comparisons between our algorithm and other algorithms. Finally, we conclude our work and discuss the future research directions in Section 5.

## 2. Related works

Interactive shape modification or editing is one of the primary operations in geometric modeling and computer animation. Space deformation is an efficient solution to this task. Barr [4] first proposed a global and local deformation method, which can stretch, bend, twist, or taper a shape. The deformations can be expressed as analytical functions of space position, such as translation, rotation, scaling, or their combinations. However, it is not intuitive for end users due to the indirect connection between control parameters and the deformation result. In 1986, Sederberg and Parry [5] proposed the classical space deformation approach, namely Free-Form Deformation (FFD). In FFD, the object is first embedded into an intermediate space, *e.g.* a

* Corresponding author. Tel.: +86 571 8820 6681.
*E-mail address:* jqfeng@cad.zju.edu.cn (J. Feng).

0097-8493/$ - see front matter © 2012 Elsevier Ltd. All rights reserved.
http://dx.doi.org/10.1016/j.cag.2012.12.001

trivariate Bézier volume [5]. Then users deform the intermediate space by editing the control points of the Bézier volume. Finally the deformation of the intermediate space is transformed to the embedded object. Due to its intuitiveness and efficiency on shape modification and editing, FFD is widely investigated and extended from various aspects, such as definitions of intermediate space, interactive means of deformation manipulations, *etc.* In 2008, Gain and Bechmann gave a detailed survey on space deformation methods [6].

Among FFD and its extensions, the deformation is conducted on sampling points of the object. Thus the deformation result will be influenced by the sampling rate or frequency, especially for the polygonal object, which is the prevalent geometric representation in computer animation. To address the problem, uniform up-sampling of the object is a straightforward solution. As a result, many tiny triangles may be generated, which will be a heavy burden for the subsequent interactive deformation and rendering. Other solutions are adaptive sampling approaches [7–9], which take the triangle size, curvature, *etc.*, of the deformed object into account. As we know, there are always singular cases which cannot be covered by the above adaptive approaches.

FFD can generate both polynomial approximation and analytical expression of the object. Some CAD/CAM systems need the latter. If the object and the deformation are represented in terms of polynomials, so is the result. But the degree of the resulting polynomial is too high to be admitted by some CAD systems. Sánchez [10] proposed an algorithm in order to reduce the degree of the result by Hermite approximation.

Feng et al. [1,2,11] proposed Accurate FFD to solve the sampling problem from the point of view of functional composition between a planar polygon and a trivariate Bernstein polynomial. Unlike the point-wise deformation approaches, they deformed polygons as trimmed Bézier surfaces. The deformation result is accurate and free of sampling problems at the price of heavy computational costs. Thus interactive Accurate FFD operation is almost prohibited. However, the interactive manipulation is the primary requirement for shape deformation in computer animation and geometric modeling. Thus how to address the problem of expensive computational cost in Accurate FFD is important.

After careful analysis of Accurate FFD algorithm, there are three main steps which contain large amount of computations. They are computations of control points of trimmed Bézier surfaces, tessellation of trimmed Bézier surfaces and rendering of trimmed Bézier surfaces. The first step is totally implemented on CPU, which cannot afford real-time computation for our case. The tessellations of trimmed Bézier surfaces include a lot of Bézier surface evaluations, which are also executed on CPU. Furthermore, all of the tessellated triangles are rendered by a vertex-at-once approach. All of the above operations will result in very low performance for Accurate FFD. In fact, the functions related to NURBS in OpenGL were removed through its deprecation mechanism since OpenGL 3.1 [12]. Thanks to more flexible parallel computing power provided by the contemporary GPGPU, the above computations can be ported into GPU, which will make interactive Accurate FFD possible.

Because of the heavy computational cost of FFD and rapid development of GPU, some GPU-based FFD acceleration methods have already been proposed in recent years. Chua [13] designed a new GPU architecture to accelerate the EFFD (Extended Free-Form Deformation) algorithm [14]. However, neither AMD nor NVIDIA has released any hardware for this particular purpose. Instead, the GPU vendors chose another way to develop their products: general purpose computing. The computing capability of GPU is more and more powerful allowing FFD to be implemented on GPU. Schein [15] presented a method for hardware-based FFD evaluation by which complex models can be deformed interactively. The method is implemented in Cg [16], thus the author adopted a lot of subtle skills to overcome the limitations of the graphics pipeline. Hahmann et al. [17] presented an approach to find an explicit closed-form solution for volume-preserving FFD by taking advantage of the multi-linear property of the volume constraint. It is the first time for them to present a GPU implementation of volume-preserving FFD. In the domain of biomedicine, Modat [18,19] proposed a fast FFD method using GPU for the non-rigid registration of MRI. None of the approaches mentioned above can be applied to Accurate FFD.

## 3. GPU acceleration of Accurate FFD *via* CUDA

### 3.1. Overview of the proposed algorithm

The proposed algorithm is GPU acceleration of Accurate FFD. Before the overview of the proposed algorithm, we will introduce the Accurate FFD [11] briefly with the aid of Fig. 1. First, the input is a polygonal object as shown in Fig. 1(a). Second, the initial B-spline volume is defined according to the bounding box of the object, user specified degrees and number of control points of B-spline. As a result, the control lattices (Fig. 1(b)) and the knot boxes (Fig. 1(c)) are generated. The object is embedded into the B-spline volume linearly. Third, each face of the object is subdivided or clipped against a knot box so that each subdivided face lies in a knot box as shown in Fig. 1(d). Fourth, the coplanar subdivided faces in a knot box are marked and will be deformed as trimmed patches of a Bézier surface. The coplanar faces are displayed in the same color in Fig. 1(e). Next, the subdivided object is deformed by the initial B-spline volume. The object remains unchanged since the initial B-spline volume is linear, as shown in Fig. 1(f). Finally, when users modify the control points of the B-spline, *i.e.*, conducting FFD, the polygonal object is deformed as curved object in terms of trimmed Bézier surface patches (Fig. 1(g)).

The proposed algorithm contains the same steps as the Accurate FFD [11], its flowchart is shown in Fig. 2, where GPU accelerating steps are indicated. Originally, all of the steps are executed on CPU except for the last step, *i.e.* rendering of tessellated trimmed Bézier surfaces on GPU, which will transfer a large amount of mesh data and Bézier surface data from CPU to GPU *via* the PCI bus.

In the proposed algorithm, the first two steps of inputting the polygonal object and initializing B-spline volume steps can only be accomplished on CPU since GPU does not provide any interactive interfaces. The successive three steps, *i.e.*, polygonal object subdivision, marking coplanar polygons, and determination of trimmed surfaces' degrees, are relatively complex and somewhat time-consuming. However, they are computed only once after the polygonal object is loaded and the B-spline volume is determined. Moreover, it is more convenient and efficient to implement them on CPU than on GPU because they contain many memory allocations and de-allocations.

The last three steps are extremely time-consuming. They are the actual bottlenecks of interactive Accurate FFD. Computing control points of the trimmed Bézier surfaces mainly involves a large number of B-spline volume evaluations, matrix and vector multiplications. Tessellating the trimmed Bézier surfaces also contains a large number of Bézier surface evaluations. Fortunately, all of the computations in these two steps can be abstracted as matrix and vector multiplications. Furthermore, these computations can be fully parallel processed. Since the tessellation is performed on GPU, the geometric data transfer in rendering the tessellated trimmed surfaces is avoided. With the help of vertex buffer objects, the Accurate FFD result can be efficiently rendered on GPU.

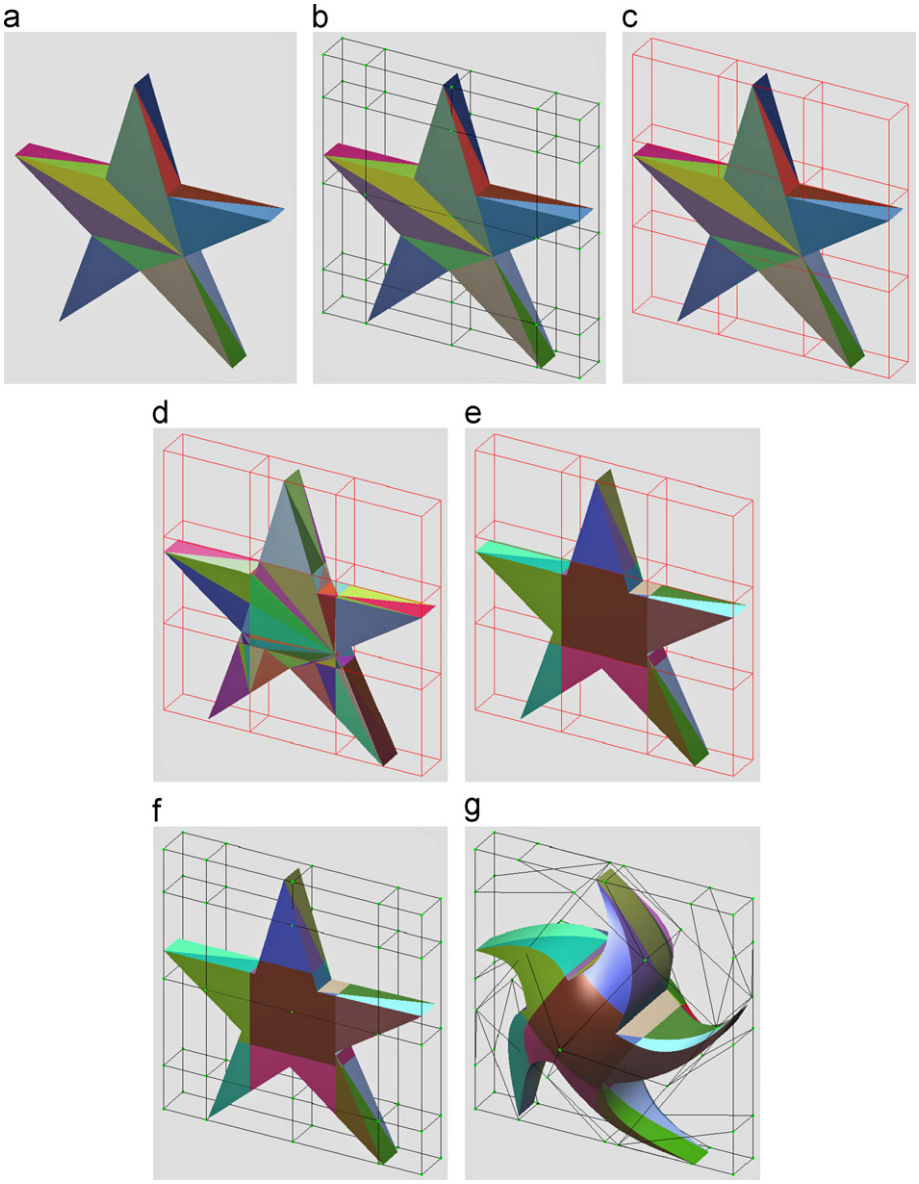


**Fig. 1.** Details of Accurate FFD.

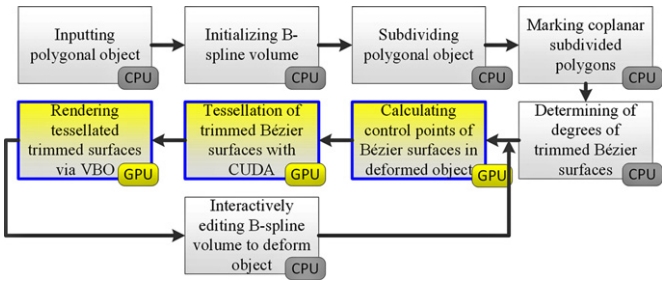


**Fig. 2.** Flow chart of B-spline FFD *via* GPU acceleration.

In this paper, we will inherit all the notations in the paper [11]. Here we briefly introduce some notations and formulae. Let $\mathbf{R}(u,v,w)$ be a B-spline volume of degree $k_u \times k_v \times k_w$

$$\mathbf{R}(u,v,w) = \sum_{i=0}^{n_u-1}\sum_{j=0}^{n_v-1}\sum_{k=0}^{n_w-1} \mathbf{R}_{ijk} N_{i,k_u}(u) N_{j,k_v}(v) N_{k,k_w}(w) \quad (1)$$

Its knot vectors are $\{u_i\}_{i=0}^{n_u+k_u}$, $\{v_j\}_{j=0}^{n_v+k_v}$ and $\{w_k\}_{k=0}^{n_w+k_w}$, respectively. First, the input polygonal object is subdivided against the B-spline knot vectors so that each of the generated sub-polygons lies in a knot box. Second, the coplanar sub-polygons are marked and the degree of final Bézier surface is determined for each knot box and each coplanar sub-polygons. Third, a 2D rectangular bounding box is computed for each coplanar sub-polygon. For each knot box $[u_i,u_{i+1}) \times [v_j,v_{j+1}) \times [w_k,w_{k+1})(i \in [0,n_u+k_u-1], j \in [0,n_v+k_v-1], k \in [0,n_w+k_w-1])$ of the B-spline volume $\mathbf{R}(u,v,w)$, there may be several planar rectangular bounding boxes in it. Each of the rectangles will be deformed as a Bézier surface $\mathbf{P}(s,t)$ of degree $n_s \times n_t$, whose control points are noted as $\{\mathbf{P}_{ij}\}_{i=0}^{n_s}{}_{j=0}^{n_t}$. Finally, the accurate deformation result, *i.e.*, the trimmed Bézier surface, is defined by $\mathbf{P}(s,t)$ and the boundaries of the coplanar subdivided triangles in the knot box.

### 3.2. Computing control points of Bézier surfaces in parallel via GPU

To compute the control points of Bézier surface $\mathbf{P}(s,t)$, we need to evaluate $(n_s+1)(n_t+1)$ sampling points $\{\mathbf{P}(s_i,t_j)\}_{i=0}^{n_s}{}_{j=0}^{n_t}$ *via* B-spline volume $\mathbf{R}(u,v,w)$ evaluations, and then interpolate them *via*

constant matrices for the given degree $(n_s+1)(n_t+1)$ of $\mathbf{P}(s,t)$. Rather than choosing the uniform sampling points $\{i/n_s, j/n_t\}_{i=0\ j=0}^{n_s\ n_t}$ in the domain $[0,1]\times[0,1]$ of $\mathbf{P}(s,t)$ as in the paper [11], we adopt the Chebyshev points as follows:

$$\begin{cases} s_i = \dfrac{1-\cos\left(\dfrac{2*i+1}{2*(n_s+1)}\pi\right)}{2} \\ t_j = \dfrac{1-\cos\left(\dfrac{2*j+1}{2*(n_t+1)}\pi\right)}{2} \end{cases} \quad i\in[0,n_s],\ j\in[0,n_t] \tag{2}$$

The condition number of Chebyshev interpolation matrix grows as an exponential function with base 2, which is smaller than the uniform sampling approach (2.3 or so) according to our experiments. Thus the Chebyshev point interpolation is more stable than the uniform sampling approach.

According to the analysis above, there are two-step evaluations to obtain the control points of $\mathbf{P}(s,t)$, *i.e.*, evaluating the sampling points $\{\mathbf{P}(s_i,t_j)\}_{i=0\ j=0}^{n_s\ n_t}$ from the B-spline volume $\mathbf{R}(u,v,w)$ and computing the control points of $\mathbf{P}(s,t)$ *via* directional polynomial interpolations of $\{\mathbf{P}(s_i,t_j)\}_{i=0\ j=0}^{n_s\ n_t}$. There are two methods to evaluate a B-spline volume point, *i.e.*, the de Boor and Cox algorithm and the matrix multiplication approach [20]. The first approach is numerically stable but contains much more computations than the second. Thus we adopt the matrix multiplication approach. The evaluation of the controls points of $\mathbf{P}(s,t)$ can also be accomplished *via* the matrix multiplication approach. For the given degree $k$, the reconstruction matrix $\mathbf{B}_k^{-1}$ is constant and can be pre-computed. In our paper, the degrees of B-spline volume are set as $1\le k_u,k_v,k_w\le 3$, which can generally meet the requirements of practical applications. Then the maximum degree of $\mathbf{P}(s,t)$ is less than $6\times 9$ [11]. Thus only 9 reconstruction matrices $\{\mathbf{B}_k^{-1}\}_{k=1}^{9}$ are pre-computed. The directional polynomial interpolations for control points of $\mathbf{P}(s,t)$ can simply be formulated as the following two matrix multiplications:

$$\tilde{\mathbf{P}} = \mathbf{B}_{n_s}^{-1}\mathbf{P} \tag{3}$$

$$\mathbf{P}_{cp} = \tilde{\mathbf{P}}(\mathbf{B}_{n_t}^{-1})^{\mathrm{T}} \tag{4}$$

where

$$\mathbf{P} = \begin{pmatrix} \mathbf{P}(s_0,t_0) & \mathbf{P}(s_0,t_1) & \cdots & \mathbf{P}(s_0,t_{n_t}) \\ \mathbf{P}(s_1,t_0) & \mathbf{P}(s_1,t_1) & \cdots & \mathbf{P}(s_1,t_{n_t}) \\ \vdots & \vdots & \ddots & \vdots \\ \mathbf{P}(s_{n_s},t_0) & \mathbf{P}(s_{n_s},t_1) & \cdots & \mathbf{P}(s_{n_s},t_{n_t}) \end{pmatrix} \tag{5}$$

$$\tilde{\mathbf{P}} = \begin{pmatrix} \tilde{\mathbf{P}}_{0,0} & \tilde{\mathbf{P}}_{0,1} & \cdots & \tilde{\mathbf{P}}_{0,n_t} \\ \tilde{\mathbf{P}}_{1,0} & \tilde{\mathbf{P}}_{1,1} & \cdots & \tilde{\mathbf{P}}_{1,n_t} \\ \vdots & \vdots & \ddots & \vdots \\ \tilde{\mathbf{P}}_{n_s,0} & \tilde{\mathbf{P}}_{n_s,1} & \cdots & \tilde{\mathbf{P}}_{n_s,n_t} \end{pmatrix} \tag{6}$$

$$\mathbf{P}_{cp} = \begin{pmatrix} \mathbf{P}_{0,0} & \mathbf{P}_{0,1} & \cdots & \mathbf{P}_{0,n_t} \\ \mathbf{P}_{1,0} & \mathbf{P}_{1,1} & \cdots & \mathbf{P}_{1,n_t} \\ \vdots & \vdots & \ddots & \vdots \\ \mathbf{P}_{n_s,0} & \mathbf{P}_{n_s,1} & \cdots & \mathbf{P}_{n_s,n_t} \end{pmatrix} \tag{7}$$

Because the maximum degree of $\mathbf{P}(s,t)$ is $6\times 9$, the maximum number of sampling points $\{\mathbf{P}(s_i,t_j)\}_{i=0\ j=0}^{n_s\ n_t}$ is $7\times 10$. These sampling points can be computed in parallel and share B-spline volume segment information. Similarly, the maximum number of elements in the matrices on the left side of (3) and (4) are also $7\times 10$. The elements $\{\tilde{\mathbf{P}}_{i,j}\}_{i=0\ j=0}^{n_s\ n_t}$ and $\{\mathbf{P}_{i,j}\}_{i=0\ j=0}^{n_s\ n_t}$ can also be evaluated in parallel.

In summary, for each Bézier surface in a knot box of B-spline volume, the directional polynomial interpolations for control points both consist of at most 70 parallel tasks. So the size of the thread block must be larger than 70. We choose the best size of thread blocks by using CUDA occupancy calculator [21], which is a tool published by NVIDIA to accomplish this task. In our case, the best size of the thread block is 352. However, there are some Bézier surfaces whose degrees are lower than $6\times 9$ and they will occupy much less threads. Thus it will be a waste of resources if we put such a Bézier surface into a thread block.

Our solution is feeding the Bézier surfaces into a thread block as many as possible. However, the above assignments of the thread blocks are too complicated for a thread to obtain its index of the operands *via* only block index and thread index. Here we construct a task lookup table to address this problem. The lookup table is a 2D array: its width is 352, which equals the size of the thread block, its height is the amount of the thread block. Its structure is shown in Fig. 3: boxPointIdx and surfaceIdx are both integers. The first one contains the knot box index and the sampling point index of one sampling point. These two variables are stored in one integer by bitwise operations. The second one contains the surface index of one sampling point. Thus a CUDA thread can simply obtain its task, which is one sampling point, *via* its blockIdx and threadIdx: lookupTable[blockIdx][theadIdx].

To summarize, the whole progress of the GPU parallel evaluations of Bézier surfaces control points can be described as follows:

1. Construct a task lookup table*.
2. Transfer all the information necessary to device (GPU) memory, such as the knot boxes, the task lookup table, and so on*.
3. Call a CUDA kernel to start the evaluation. Each block contains 352 threads.
4. Transfer $\mathbf{B}_{n_s}^{-1}$, $\mathbf{B}_{n_t}^{-1}$ and three matrices of B-spline volume evaluation from global memory to shared memory in parallel.
5. For each thread, obtain its task with parameters blockIdx and threadIdx from the task lookup table.
6. Each thread evaluates a B-spline volume sampling point $\mathbf{P}(s_i,t_j)(0\le i\le n_s, 0\le j\le n_t)$ and writes it to the shared memory.
7. Each thread computes a temporary control point $\tilde{\mathbf{P}}_{i,j}(0\le i\le n_s, 0\le j\le n_t)$ and writes it to the shared memory.
8. Each thread computes a control point $\mathbf{P}_{i,j}(0\le i\le n_s, 0\le j\le n_t)$ of Bézier surface $\mathbf{P}(s,t)$.

Where, asterisk (*) means that the data in this step can be computed and loaded to the GPU just once. All the other steps are executed on-the-fly.

### 3.3. GPU tessellations of trimmed Bézier surfaces

In CPU implementations, the trimmed Bézier surfaces are tessellated by calling GLU library functions directly. The tessellated triangles are then transferred to GPU for rendering. In general, each trimmed Bézier surface will be tessellated into over 400 triangles for high quality deformation. As a result, the data

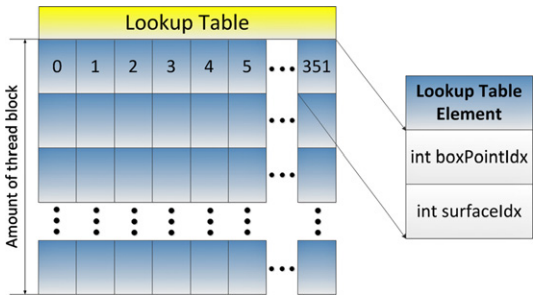


**Fig. 3.** Structure of the task lookup table.

transfer and tessellated triangles rendering are very time consuming. In the proposed GPU implementation, the trimmed Bézier surfaces are efficiently tessellated in parallel on GPU. Then the tessellated triangles are rendered directly. Thus the data transfers from main memory to graphics memory are avoided.

Recently, some GPU based Bézier tessellation algorithms are proposed. Schwarz et al. [22] first estimate tessellation factors, and then tessellate the surfaces adaptively. Finally, the resulting triangles are rendered using OpenGL vertex buffer object. This tessellation framework is based on CUDA. Concheiro et al. [23] first perform some different error tests to guide the tessellation procedure. Then they generate the tessellation patterns. Finally the tessellation triangles are generated *via* geometry shaders.

The above algorithms generate high-quality tessellation result of Bézier surfaces. Since they are adaptive approaches, it is not a trivial work to extend them to our trimmed Bézier surfaces patches case. Besides, the trimming lines of the Bézier surface patches are defined explicitly, *i.e.*, the parametric domains of the Bézier surface patches are polygons. Here we proposed a simple and intuitive tessellation approach: first uniformly tessellating the parametric domain(polygon), then compute the image of the tessellated domain on the trimmed Bézier surface patch.

In our algorithm, the input object is a triangular mesh or a polygonal mesh. It is reasonable to assume that each face in the polygonal object is convex. Otherwise, the face can be converted into convex ones by adding auxiliary lines. In the Accurate FFD algorithm [11], each polygonal face is subdivided or clipped against a rectangular box. The resulting polygonal faces are also convex. According to the Accurate FFD algorithm [11], each convex polygonal face will be deformed as a trimmed Bézier surface patch, where the boundary curve of the polygonal face is the trimming lines. To display the Bézier surface patch, it should be tessellated and fetched into the rendering pipeline. Thanks to its trimming lines defined explicitly, we first tessellate the parametric domain (*i.e.* a polygonal face) of the trimmed Bézier surface patch, then compute the image of the tessellated domain on the corresponding Bézier surface. Our tessellation pattern in the parametric domain of the trimmed Bézier surface patch is shown as Fig. 4. For each convex n-sided polygon, it is triangulated as $(n-2)$ triangles. For each triangle, its image on the surface is also a Triangular Trimmed Bézier Patch, which is abbreviated as TTBP, and it is uniformly subdivided into $m^2$ sub-triangles, where $m$ is the number of equal subdivision along triangle edges. Then for each tessellated point in the tessellated domain, we compute its corresponding point and normal on the Bézier surface $\mathbf{P}(s,t)$. The point and normal evaluations will be accomplished *via* Bézier matrix multiplications, rather than the de Casteljau algorithm, for the same reason as the GPU evaluation of B-spline volume—the de Casteljau algorithm contains more computations than the matrix multiplication approach.

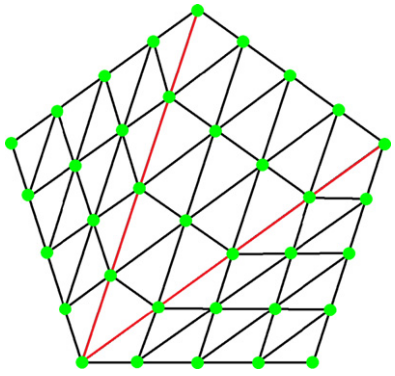

**Fig. 4.** Tessellation pattern of parametric domains of trimmed Bézier surface patches.

There are two advantages of adopting the above simple tessellation approach. First, due to the same tessellation resolution, each adjacent edge of any two sub-triangles has the same number of sampling points. Thus the tessellated trimmed Bézier surfaces are watertight, which will be free of cracks among different trimmed Bézier surface patches. Second, for each trimmed Bézier surface, the number of sampling points can be computed accurately. For example, for an $n$-side trimmed patch with tessellation resolution $m$, the tessellated trimmed surface has $(n-2)m^2$ triangles and $(n-2)(m+1)(m+2)/2$ tessellated points. Consequently, the size of GPU task lookup table is small because it is easy to assign the thread tasks.

In our implementation, one thread block which contains 128 threads will tessellate one sub-polygon. If its tessellated points are less than 128, some of the threads will be idle. Otherwise some threads will evaluate more than one tessellated point. For each tessellated point, its normal will also be evaluated accordingly for high quality rendering of the trimmed Bézier surfaces. Since each thread block will process one sub-polygon, the corresponding evaluation matrices for points and normals can be loaded to shared memory. The shared matrices can greatly improve the tessellation performance. The procedure of GPU tessellation of trimmed Bézier surfaces is:

1. Construct a task lookup table.
2. Transfer the task lookup table necessary to device (GPU) memory.
3. Call a CUDA kernel to start the calculation. The thread block amount equals to the amount of sub-polygons, each block contains 128 threads.
4. For each thread block, compute the Bézier point and normal evaluation matrices and transfer them to shared memory.
5. Each thread obtains its evaluation task according to its blockIdx and threadIdx in the task lookup table.
6. Each thread evaluates a tessellated point and its normal.

### 3.4. Optimizations of the algorithm

The algorithm above can be optimized further. The evaluation of the control points of the Bézier surfaces can be simplified by matrix manipulation. The task arrangement of the trimmed Bézier surfaces' tessellation can be redesigned for better performance. The details of optimization are described as follows.

First, from formula (3) and (4), we obtain

$$\mathbf{P}_{cp} = \mathbf{B}_{n_s}^{-1}\mathbf{P}(\mathbf{B}_{n_t}^{-1})^{\mathrm{T}} \tag{8}$$

where

$$\mathbf{B}_{n_s} = \begin{pmatrix} \binom{n_s}{0}(1-s_0)^{n_s} & \binom{n_s}{1}s_0(1-s_0)^{n_s-1} & \cdots & \binom{n_s}{n_s}s_0^{n_s} \\ \binom{n_s}{0}(1-s_1)^{n_s} & \binom{n_s}{1}s_0(1-s_1)^{n_s-1} & \cdots & \binom{n_s}{n_s}s_1^{n_s} \\ \vdots & \vdots & \ddots & \vdots \\ \binom{n_s}{0}(1-s_{n_s})^{n_s} & \binom{n_s}{1}s_0(1-s_{n_s})^{n_s-1} & \cdots & \binom{n_s}{n_s}s_{n_s}^{n_s} \end{pmatrix} \tag{9}$$

$$\mathbf{B}_{n_t} = \begin{pmatrix} \binom{n_t}{0}(1-t_0)^{n_t} & \binom{n_t}{1}t_0(1-t_0)^{n_t-1} & \cdots & \binom{n_t}{n_t}t_0^{n_t} \\ \binom{n_t}{0}(1-t_1)^{n_t} & \binom{n_t}{1}t_0(1-t_1)^{n_t-1} & \cdots & \binom{n_t}{n_t}t_1^{n_t} \\ \vdots & \vdots & \ddots & \vdots \\ \binom{n_t}{0}(1-t_{n_t})^{n_t} & \binom{n_t}{1}t_0(1-t_{n_t})^{n_t-1} & \cdots & \binom{n_t}{n_t}t_{n_t}^{n_t} \end{pmatrix} \tag{10}$$

So, $\mathbf{B}_{n_s}$ can be factored as follows:

$$\mathbf{B}_{n_s} = \mathbf{S}_{n_s}\mathbf{D}_{n_s} \tag{11}$$

where

$$\mathbf{S}_{n_s} = \begin{pmatrix} (1-s_0)^{n_s} & s_0(1-s_0)^{n_s-1} & \cdots & s_0^{n_s} \\ (1-s_1)^{n_s} & s_1(1-s_1)^{n_s-1} & \cdots & s_1^{n_s} \\ \vdots & \vdots & \ddots & \vdots \\ (1-s_{n_s})^{n_s} & s_{n_s}(1-s_{n_s})^{n_s-1} & \cdots & s_{n_s}^{n_s} \end{pmatrix} \tag{12}$$

$\mathbf{D}_{n_s}$ is a diagonal matrix:

$$\begin{pmatrix} \binom{n_s}{0} & 0 & \cdots & 0 \\ 0 & \binom{n_s}{1} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & \binom{n_s}{n_s} \end{pmatrix} \tag{13}$$

Similarly, $\mathbf{B}_{n_t}$ can be factored as follows:

$$\mathbf{B}_{n_t} = \mathbf{T}_{n_t}\mathbf{D}_{n_t} \tag{14}$$

By substituting (11) and (14) into (8), we obtain

$$\begin{aligned} \mathbf{P}_{cp} &= \mathbf{B}_{n_s}^{-1}\mathbf{P}(\mathbf{B}_{n_t}^{-1})^{\mathrm{T}} \\ &= (\mathbf{S}_{n_s}\mathbf{D}_{n_s})^{-1}\mathbf{P}[(\mathbf{T}_{n_t}\mathbf{D}_{n_t})^{-1}]^{\mathrm{T}} \\ &= \mathbf{D}_{n_s}^{-1}\mathbf{S}_{n_s}^{-1}\mathbf{P}(\mathbf{D}_{n_t}^{-1}\mathbf{T}_{n_t}^{-1})^{\mathrm{T}} \\ &= \mathbf{D}_{n_s}^{-1}\mathbf{S}_{n_s}^{-1}\mathbf{P}(\mathbf{T}_{n_t}^{\mathrm{T}})^{-1}(\mathbf{D}_{n_t}^{\mathrm{T}})^{-1} \end{aligned} \tag{15}$$

Second, the tessellated points of the trimmed Bézier surfaces can be evaluated using the following formula:

$$\mathbf{P}(s_l,t_l) = \mathbf{S}_l\mathbf{D}_{n_s}\mathbf{P}_{cp}\mathbf{D}_{n_t}^{\mathrm{T}}\mathbf{T}_l^{\mathrm{T}}, \quad 1 \le l \le q \tag{16}$$

Where $(s_l,t_l)$ is the parameter of the tessellated point, $q=(m+1)(m+2)/2$ is the number of tessellated points, $m$ is the tessellation resolution.

$$\mathbf{S}_l = [(1-s_l)^{n_s} \quad s_l(1-s_l)^{n_s-1} \quad \cdots \quad s_l^{n_s}], \quad 1 \le l \le q \tag{17}$$

$$\mathbf{T}_l = [(1-t_l)^{n_t} \quad t_l(1-t_l)^{n_t-1} \quad \cdots \quad t_l^{n_t}], \quad 1 \le l \le q \tag{18}$$

By substituting (15) into (16), we obtain

$$\begin{aligned} \mathbf{P}(s_l,t_l) &= \mathbf{S}_l\mathbf{D}_{n_s}\mathbf{D}_{n_s}^{-1}\mathbf{S}_{n_s}^{-1}\mathbf{P}(\mathbf{T}_{n_t}^{\mathrm{T}})^{-1}(\mathbf{D}_{n_t}^{\mathrm{T}})^{-1}\mathbf{D}_{n_t}^{\mathrm{T}}\mathbf{T}_l^{\mathrm{T}} \\ &= \mathbf{S}_l\mathbf{S}_{n_s}^{-1}\mathbf{P}(\mathbf{T}_{n_t}^{\mathrm{T}})^{-1}\mathbf{T}_l^{\mathrm{T}}, \quad 1 \le l \le q \end{aligned} \tag{19}$$

Thus the formula $\mathbf{B}_{n_s}^{-1}\mathbf{P}(\mathbf{B}_{n_t}^{\mathrm{T}})^{-1}$ of computing the Bézier surfaces control points described in Section 3.2 can be rewritten as $\mathbf{S}_{n_s}^{-1}\mathbf{P}(\mathbf{T}_{n_t}^{\mathrm{T}})^{-1}$. Just like $\{\mathbf{B}_k^{-1}\}_{k=1}^{9}$, the matrices $\{\mathbf{S}_k^{-1}\}_{k=1}^{9}$ and $\{\mathbf{T}_k^{-1}\}_{k=1}^{9}$ can be pre-computed. The dimensions of $\mathbf{B}_{n_s}^{-1}$ and $\mathbf{S}_{n_s}^{-1}$ are the same; the dimensions of $(\mathbf{B}_{n_t}^{-1})^{\mathrm{T}}$ and $(\mathbf{T}_{n_t}^{-1})^{\mathrm{T}}$ are the same too. In brief, the dimensions of all three matrices are unchanged. Thus, the optimized algorithm is almost the same with the algorithm in Section 3.2 except that $\mathbf{S}_{n_s}^{-1}$ replaces $\mathbf{B}_{n_s}^{-1}$ and $(\mathbf{T}_{n_t}^{\mathrm{T}})^{-1}$ replaces $(\mathbf{B}_{n_t}^{\mathrm{T}})^{-1}$.

Similarly, the tangent vector in $s$ direction is

$$\mathbf{P}_u(s_l,t_l) = \frac{\partial \mathbf{S}_l}{\partial s}\mathbf{S}_{n_s}^{-1}\mathbf{P}(\mathbf{T}_{n_t}^{\mathrm{T}})^{-1}\mathbf{T}_l^{\mathrm{T}}, \quad 1 \le l \le q \tag{20}$$

The tangent vector in $t$ direction is

$$\mathbf{P}_v(s_l,t_l) = \mathbf{S}_l\mathbf{S}_{n_s}^{-1}\mathbf{P}(\mathbf{T}_{n_t}^{\mathrm{T}})^{-1}\frac{\partial \mathbf{T}_l^{\mathrm{T}}}{\partial t}, \quad 1 \le l \le q \tag{21}$$

The normal of the tessellated point is

$$\mathbf{N}(s_l,t_l) = \mathbf{P}_u(s_l,t_l) \times \mathbf{P}_v(s_l,t_l), \quad 1 \le l \le q \tag{22}$$

If we adopted the algorithm in Section 3.3 to tessellate the trimmed Bézier surfaces, the evaluation formula is

$$\mathbf{R}(s_l,t_l)_{origin} = \mathbf{S}_l\mathbf{D}_{n_s}\mathbf{B}_{n_s}^{-1}\mathbf{P}(\mathbf{B}_{n_t}^{-1})^{\mathrm{T}}\mathbf{D}_{n_t}^{\mathrm{T}}\mathbf{T}_l^{\mathrm{T}}, \quad 1 \le l \le q \tag{23}$$

It contains six matrix multiplications, while the optimized formula (19) contains only four matrix multiplications. The matrices for computing the tessellated points and normals from Bézier surface definition can be omitted. The costs of transferring them to shared memory for each thread block are also saved, which is the 4th step in the tessellation algorithm. What's more, there is a drawback in the algorithm described in Section 3.3: one thread block which contains 128 threads handles one Bézier surface. If the number of the tessellated points is much less than 128, most of the threads will be wasted. While according to the optimized formula (19), we can rearrange the tasks in the following way: each thread handles one tessellated point no matter which surface it belongs to. The new arrangement will make full use of threads in each block. Fig. 5 shows the efficiency difference between these two algorithms. Each color block shows the relative position of the working threads in the CUDA thread block. TTBP means Triangular Trimmed Bézier Patch mentioned in Section 3.3. Fig. 6 shows the input and output of this Bézier surface tessellation kernel. Our implementation results show that the optimized algorithm is faster one fold than the unoptimized one in Section 3.3. Thus the optimized algorithm becomes more efficient.

In formula (19), we represent the matrices $\mathbf{S}_{n_s}^{-1}$ and $(\mathbf{T}_{n_t}^{\mathrm{T}})^{-1}$ in terms of the Bernstein basis except for the combinatorial numbers. This helps to improve the numerical stability of the algorithm. If the power basis is adopted in the optimized algorithm (19), the condition numbers of $\{\mathbf{S}_k^{-1}\}_{k=1}^{9}$ would be very large. As a result, the numerical error in the tessellated point will be so large that the tessellated triangles will not be watertight. According to our test, the condition number of $\{\mathbf{S}_k^{-1}\}_{k=1}^{9}$ in terms of the

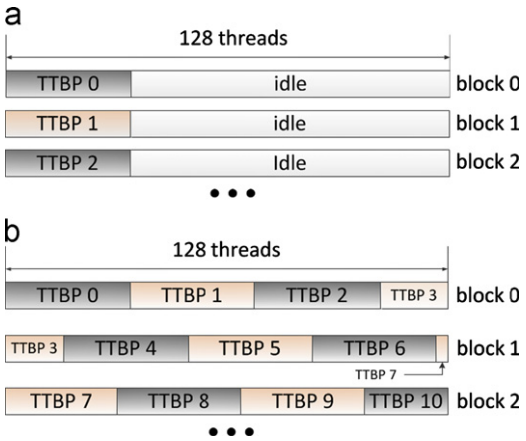


**Fig. 5.** The efficiency comparison between the original algorithm and the optimized one. (a) Original algorithm. (b) Optimized algorithm.

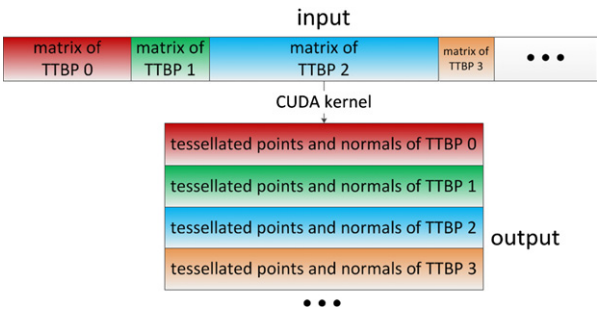


**Fig. 6.** The inputs and outputs of the Bézier surface tessellation kernel.

Bernstein basis is much smaller than that of the power basis. The comparison of condition numbers between power basis and Bernstein basis is shown in Table 1. In this way, the numerical error problem can be overcome and watertight tessellation result can be obtained.

### 3.5. Rendering the generated triangles using vertex buffer object

After the tessellation step above, all the tessellated vertices, normals and their topological connectivity are stored in the graphics memory in terms of CUDA result. It is an efficient solution to map the OpenGL buffer objects into the address space of our CUDA result, and then render the tessellated trimmed Bézier surfaces in terms of vertex buffer object [24]. The rendering approach provides very efficient performance because the geometry data is stored in the graphics memory and does not need to be transferred from main memory to graphics memory *via* PCI bus.

**Table 1**
Comparison of condition numbers of matrices $\mathbf{S}_{n_s}^{-1}$ and $(\mathbf{T}_{n_t}^{\mathrm{T}})^{-1}$.

| Matrix size | Condition number | |
|---|---|---|
| | Power basis | Bernstein basis |
| $2\times2$ | 3.61 | 1.41 |
| $3\times3$ | 19.65 | 4.92 |
| $4\times4$ | 110.25 | 14.56 |
| $5\times5$ | 626.71 | 50.84 |
| $6\times6$ | 3588.06 | 180.94 |
| $7\times7$ | 20,633.22 | 661.73 |
| $8\times8$ | 118,998.85 | 2451.94 |
| $9\times9$ | 687,713.37 | 9181.65 |

## 4. Implementation results and discussion

### 4.1. Comparison with the CPU algorithm

We implemented the proposed algorithm on a PC with Intel Core i5 760@2.8 GHz, 4 GB Memory and NVIDIA GeForce GTX 465 GPU. The operating system is openSUSE 12.1. The CPU part of our algorithm is implemented by C++ and the GPU part is implemented by CUDA C. We also integrated the direct manipulation of FFD algorithm [25] into our system. Some examples are shown in Figs. 7–11, among which Fig. 9 is an example of direct manipulation of FFD. In Fig. 10, the difference between FFD result and our result is inconspicuous because the size of each face of the original model is small, the aliasing problem in the deformation is not so serious. However, the differences will be manifested if we zoom the model and observe the geometric details. As an example, the detailed CPU and GPU run time comparisons of the model "chair" in Fig. 9 are given in Tables 2 and 3.

Although the proposed GPU acceleration algorithm requires slightly overhead in the preprocessing step, the user interaction is much faster than the CPU algorithm. To compute the matrix $\mathbf{S}_{n_s}^{-1}\mathbf{P}(\mathbf{T}_{n_t}^{\mathrm{T}})^{-1}$, the GPU algorithm is more than 50 times faster than the CPU algorithm. In the trimmed surfaces tessellations and rendering steps, the GPU algorithm is about 200 times faster than the CPU algorithm. Even for a simple model of 1536 triangles in Fig. 7, it is almost impossible to achieve interactive deformation

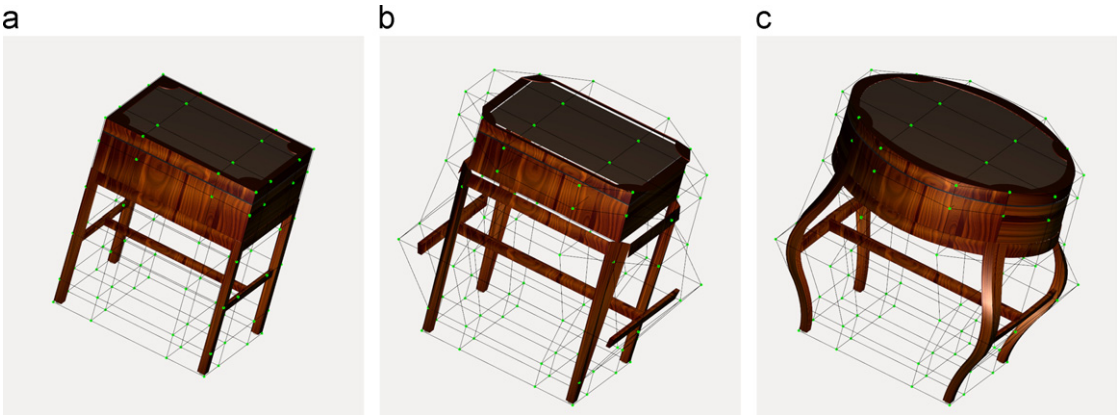


**Fig. 7.** A square seat and its deformation results. (a) Original model, (b) FFD result and (c) Our result.

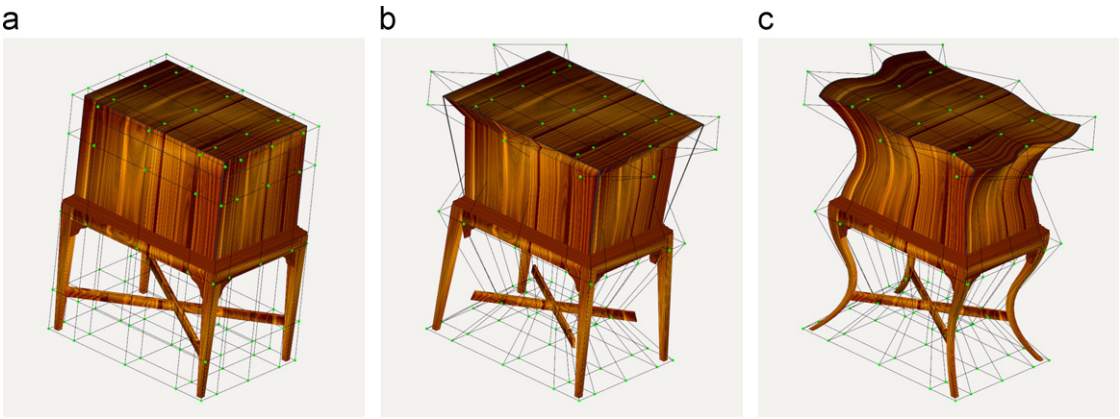


**Fig. 8.** A side table and its deformation results. (a) Original model, (b) FFD result and (c) Our result.

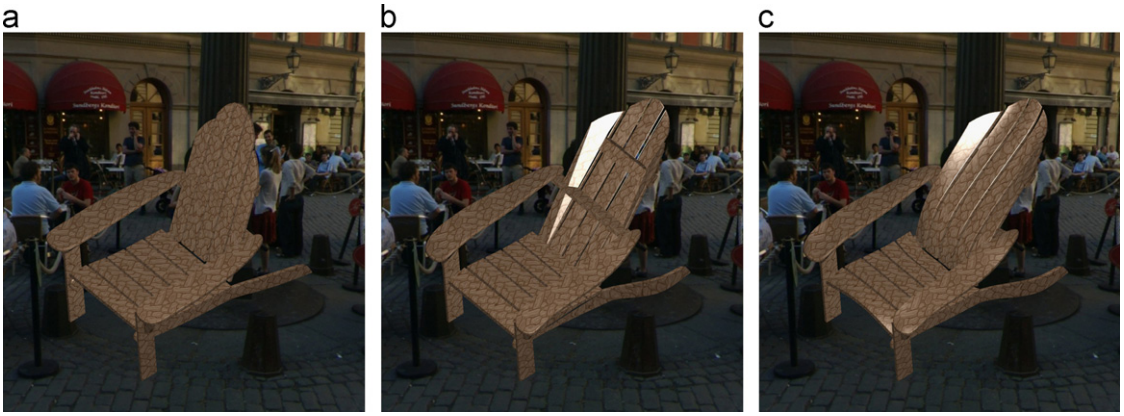


**Fig. 9.** A chair and its deformation results. (a) Original model, (b) FFD result and (c) Our result.

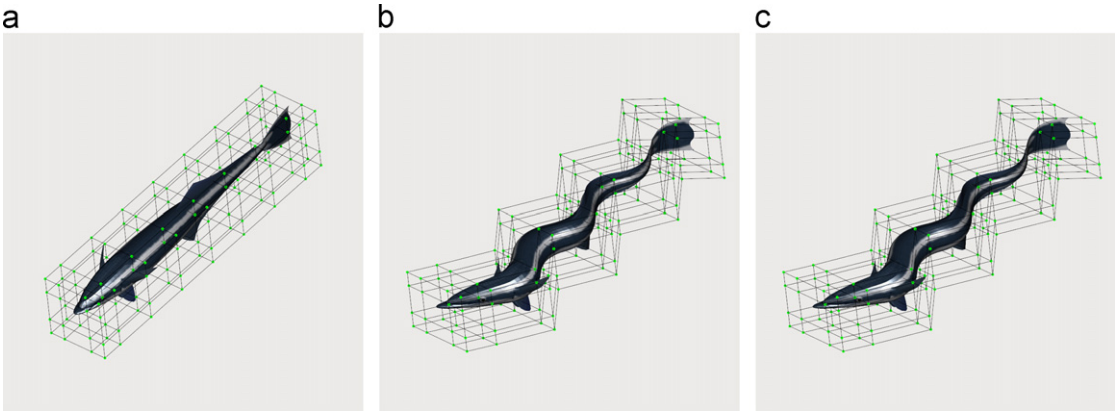


**Fig. 10.** A fish and its deformation results. (a) Original model, (b) FFD result and (c) Our result.

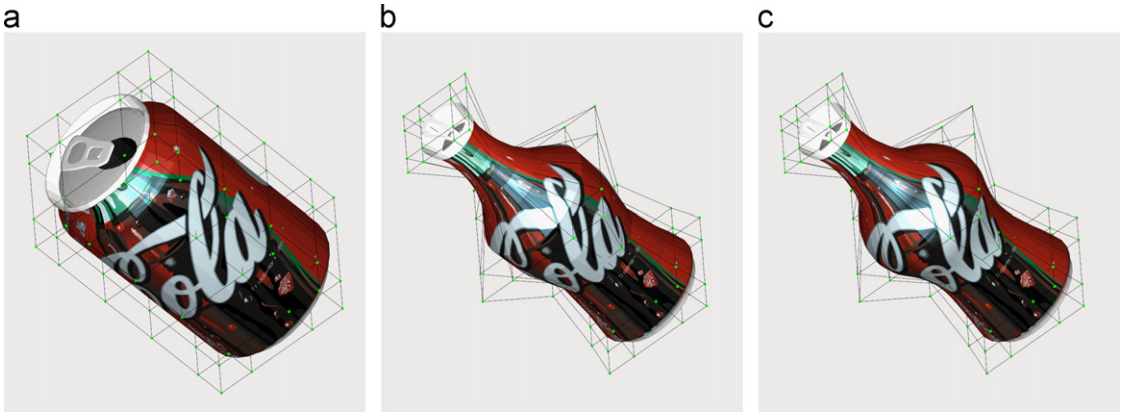


**Fig. 11.** A coke tin and its deformation results. (a) Original model, (b) FFD result and (c) Our result.

with CPU. But the GPU algorithm can deform the coke model in Fig. 11 at above 20 fps, which contains 14,153 triangles.

### 4.2. Comparison with the tessellation shader algorithm

Tessellation shaders have been added in OpenGL since version 4.0. Of course, the trimmed Bézier can be tessellated *via* tessellation shaders. For the sake of comparison, we implement three trimmed Bézier surface tessellation algorithms using tessellation shaders. In the 1st algorithm, we evaluate every sampling point and its normal by the de Casteljau algorithm in the tessellation evaluation shader. In the 2nd algorithm, we evaluate every sampling point and its normal by matrix multiplication in the tessellation evaluation shader. In the 3rd algorithm, we first calculate the middle three matrices of Eq. (19) by CUDA and store them. In this way, we can calculate the sampling points and their normals in the tessellation evaluation shader by fewer matrix multiplications.The detailed runtime statistics are given in Table 4. It indicates that the proposed CUDA algorithm is much faster than the three tessellation shader algorithms.

### 4.3. Comparison with the uniformly up-sampling algorithm

As described in Section 2, we can obtain the same rendering result of Accurate FFD by uniformly up-sampling the polygonal

**Table 2**
Loading time statistics of model "chair" in Fig. 9.

| Parameter | GPU time (ms) | CPU time (ms) |
|---|---|---|
| Load model | | 31 |
| Allocate memory | | 38 |
| Define initial B-spline volume: degree $3\times3\times3$, control points $9\times9\times9$ | | $<1$ |
| Copy knot vectors and control points from main memory to GPU memory | $<1$ | – |
| Subdivide model according to knot vectors | | 191 |
| Mark coplanar sub-polygons | | 12 |
| Calculate coplanar sub-polygons' bounding boxes | | 4 |
| Generate a GPU task list for calculating $\mathbf{S}_{n_s}^{-1}\mathbf{P}(\mathbf{T}_{n_t}^{\mathrm{T}})^{-1}$ | 2 | – |
| Load Bézier surfaces' information to GPU memory | 5 | – |
| Load the GPU task list of $\mathbf{S}_{n_s}^{-1}\mathbf{P}(\mathbf{T}_{n_t}^{\mathrm{T}})^{-1}$ calculation to GPU memory | 102 | – |
| Generate a GPU task list for tessellating the trimmed Bézier surfaces | 16 | – |
| Load the subdivision results to GPU memory | 7 | – |
| Total | 408 | 276 |

**Table 3**
Runtime statistics of interactive Accurate FFD (ms).

| Model (triangles) | Seat (1536) | | Table (2336) | | Chair (3878) | | Fish (7066) | | Coke (14,513) | |
|---|---|---|---|---|---|---|---|---|---|---|
| Computing device | CPU | GPU | CPU | GPU | CPU | GPU | CPU | GPU | CPU | GPU |
| Edit control points | $<1$ | $<1$ | $<1$ | $<1$ | $<1$ | $<1$ | $<1$ | $<1$ | $<1$ | $<1$ |
| Copy control points to GPU | – | $<1$ | – | $<1$ | – | $<1$ | – | 1 | – | 1 |
| Calculate $\mathbf{S}_{n_s}^{-1}\mathbf{P}(\mathbf{T}_{n_t}^{\mathrm{T}})^{-1}$ | 66 | 1 | 80 | 1 | 235 | 4 | 457 | 7 | 1073 | 15 |
| Evaluate sampling points and normals | – | 2 | – | 4 | – | 6 | – | 9 | – | 24 |
| Render the deformed object | 710 | 1 | 1271 | 2 | 1798 | 2 | 2598 | 4 | 5850 | 8 |
| Total | 776 | 4 | 1351 | 7 | 2033 | 12 | 3451 | 21 | 4193 | 48 |

**Table 4**
Runtime comparisons between CUDA and shader algorithms (ms).

| Model (triangles) | Seat (1536) | Table (2336) | Chair (3878) | Fish (7066) | Coke (14,513) |
|---|---|---|---|---|---|
| 1st algorithm | 58 | 79 | 196 | 355 | 1016 |
| 2nd algorithm | 38 | 56 | 109 | 185 | 497 |
| 3rd algorithm | 16 | 23 | 47 | 79 | 203 |
| CUDA | 4 | 7 | 12 | 21 | 48 |

object and deforming these up-sampling points directly. In this up-sampling approach, the deformation result is triangular meshes, rather than analytical trimmed Bézier patches. Obviously, the algorithm can also be implemented *via* CUDA, which is called UUS algorithm for short. All of the sampling points in the UUS algorithm are computed *via* B-spline volume evaluations. However, the sampling points in our algorithm are computed *via* Bézier surface evaluations. It is reasonable to guess that the proposed algorithm is more efficient than the UUS algorithm because the Bézier surface is defined *via* a bivariate function while the B-spline volume is *via* a trivariate one. However, there is an overhead in the proposed algorithm: reconstruction of the Bézier surfaces. Thus when the number of the sampling points is small, our algorithm maybe slower than the UUS algorithm. But if the number of the sampling points becomes larger, the evaluations of the sampling points will dominate the computational time. Our algorithm will manifest its efficiency. In what follows, we will analyze the complexities of UUS algorithm and the proposed algorithm, especially when the number of the sampling points varies. Assume that the polygonal object will be deformed as $M$ trimmed Bézier patches. Each patch will be tessellated by $P$ up-sampling points. The B-spline volume's degree is $k_u\times k_v\times k_w$, so its order is $r_u\times r_v\times r_w=(k_u+1)\times(k_v+1)\times(k_w+1)$.

So, the time complexity of evaluating one point in the B-spline volume in terms of MAD operation is

$$T=(r_w^2+3r_ur_vr_w+r_v^2+3r_ur_v+r_u^2+3r_u) \tag{24}$$

Where a MAD operation means an arithmetic multiply and add operation. In the UUS algorithm, all of calculations are the evaluations of $3MP$ points in the B-spline volume, where "3" means one deformed point and two auxiliary points for the normal estimation. Its complexity in terms of MAD operation is $3MPT$.

In the proposed Accurate FFD algorithm, we assume all the Bézier surfaces are of maximum degree $n_s\times n_t=(k_u+k_v)(k_u+k_v+k_w)$, which is the worst case [11]. First, for each Bézier surface, $(n_s+1)\times(n_t+1)$ points in the B-spline volume should be evaluated to determine its control points $\{\mathbf{P}_{i,j}\}_{i=0}^{n_s}{}_{j=0}^{n_t}$ as mentioned in Section 3.2. In fact, we do not compute the control points $\{\mathbf{P}_{i,j}\}_{i=0}^{n_s}{}_{j=0}^{n_t}$ explicitly according to the optimized algorithm in Section 3.4. Second, several trimmed Bézier patches may share one Bézier surface. Thus only evaluation of $(n_s+1)\times(n_t+1)$ B-spline volume points is necessary and the complexity of the worst case is $M(n_s+1)(n_t+1)T$. Third, for each trimmed Bézier patch, multiplication of three matrices should be implemented, which is $\mathbf{S}_{n_s}^{-1}\mathbf{P}(\mathbf{T}_{n_t}^{\mathrm{T}})^{-1}$ as mentioned in Section 3.4. Its complexity is

$$\begin{aligned}&M[(n_s+1)(n_t+1)\times(n_s+1)+(n_s+1)(n_t+1)\times(n_t+1)]\\&\quad=M[(n_s+1)(n_t+1)(n_s+n_t+2)]\end{aligned} \tag{25}$$

Finally, for each trimmed Bézier patch, the sampling points and normals on the Bézier surface should be evaluated. It is accomplished by matrix multiplications as mentioned in Section 3.4: $\mathbf{S}_l\mathbf{S}_{n_s}^{-1}\mathbf{P}(\mathbf{T}_{n_t}^{\mathrm{T}})^{-1}\mathbf{T}_l^{\mathrm{T}}$, where $\mathbf{S}_{n_s}^{-1}\mathbf{P}(\mathbf{T}_{n_t}^{\mathrm{T}})^{-1}$ has been finished in the second step. The normal vector is the cross-product of two tangent vectors, which can be evaluated by the same way. Thus

its complexity is

$$3MP[(n_s+1)(n_t+1)+(n_t+1)] = 3MP(n_s+2)(n_t+1) \quad (26)$$

For simplicity, we assume that $k_u = k_v = k_w = k$. The complexity comparison in terms of numbers of MAD operations between the UUS algorithm and the proposed Accurate FFD algorithm are given in Table 5 and Fig. 12.

Fig. 12 shows the time complexity comparison of the two algorithms intuitively.

From the theoretical comparison above, we can conclude that the Accurate FFD algorithm is faster than the UUS one in the case of rendering the deformation result. Furthermore, the representation of the deformation result of the Accurate FFD, *i.e.*, trimmed Bézier patches, is more compact than those of the UUS algorithm, *i.e.*, densely sampled triangular meshes. In the UUS algorithm, all of the sampling points on the deformed object are obtained *via* the B-spline volume evaluations. In the proposed Accurate FFD algorithm, all of the sampling points on the deformed object are obtained *via* the Bézier surface evaluations, which is cheaper than the B-spline volume evaluations. In addition, there are a lot of matrix operations in the Accurate FFD algorithm. These matrices can be shared among threads for the sake of high performance. But the shared data in the UUS algorithm is relatively fewer. This difference makes the performance gap between these two algorithms larger. The only disadvantage in the Accurate FFD algorithm is the overhead of the preprocessing step, *i.e.* computing the Bézier surfaces. For the rendering and interaction purposes, the preprocessing step can be omitted, which is replaced by a matrix multiplication $\mathbf{S}_{n_s}^{-1}\mathbf{P}(\mathbf{T}_{n_t}^{\mathrm{T}})^{-1}$. Our implementation results also prove the above conclusion as shown in Table 6. In general, the Accurate FFD algorithm is three times faster than the UUS ones.

**Table 5**
Complexities of UUS algorithm and Accurate FFD in terms of number of MAD operations.

| $P$ | $k=1$ | | $k=2$ | | $k=3$ | |
|---|---|---|---|---|---|---|
| | UUS | Accu FFD | UUS | Accu FFD | UUS | Accu FFD |
| 10 | 1620 | 1212 | 4320 | 6720 | 9000 | 24,590 |
| 20 | 3240 | 1692 | 8640 | 7980 | 18,000 | 26,990 |
| 30 | 4860 | 2172 | 12,960 | 9240 | 27,000 | 29,390 |
| 40 | 6480 | 2652 | 17,280 | 10,500 | 36,000 | 31,790 |
| 50 | 8100 | 3132 | 21,600 | 11,760 | 45,000 | 34,190 |
| 60 | 9720 | 3612 | 25,920 | 13,020 | 54,000 | 36,590 |
| 70 | 11,340 | 4092 | 30,240 | 14,280 | 63,000 | 38,990 |
| 80 | 12,960 | 4572 | 34,560 | 15,540 | 72,000 | 41,390 |
| 90 | 14,580 | 5052 | 38,880 | 16,800 | 81,000 | 43,790 |
| 100 | 16,200 | 5532 | 43,200 | 18,060 | 90,000 | 46,190 |

## 5. Conclusion

In this paper, we proposed a GPU acceleration of the Accurate FFD algorithm. By carefully analyzing the algorithm, the control points of the resulting Bézier surface do not need to be computed explicitly. Furthermore, the optimized scheme to tessellate the trimmed Bézier surface is more efficient and numerically stable than the trivial one. As a result, we can achieve interactive Accurate FFD on GPU.

There are several aspects to improve the proposed GPU algorithm further. The density of tessellated points is a constant for all the trimmed Bézier patches. It is a simple solution, especially for GPU implementation. Obviously, it is an inefficient solution. If the sizes of trimmed Bézier patches are different, we should adopt the tessellation factor according to the largest patch for the sake of good visual effect. As a side-effect, too much tiny triangles may be produced, which will be the wastes of computing and memory resources. In the future, we will design an adaptive tessellation algorithm to handle this. Another potential improvement relates to the GPU. We implemented our algorithm using CUDA, which requires G80 or newer NVIDIA GPUs. It is platform-dependent. In the future, we will implement the algorithm with OpenCL or DirectCompute so that it can be applicable to both AMD and NVIDIA GPUs.

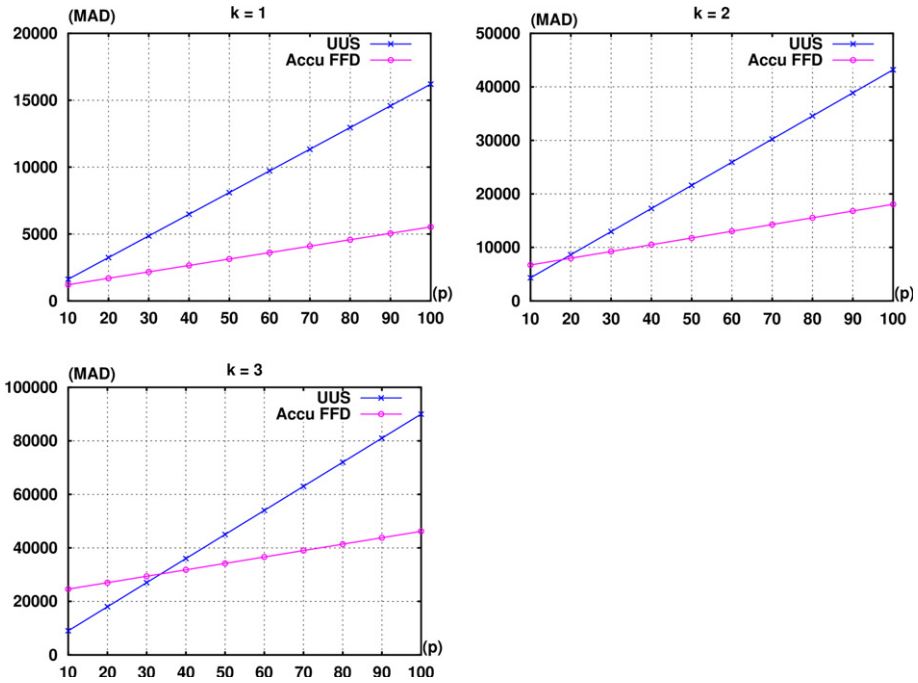


**Fig. 12.** Time complexity comparison between UUS and Accurate FFD.

**Table 6**
Implementation comparison between the Accurate FFD and the UUS algorithms.

| Model (triangles) | Table (2336) | | | | Chair (3878) | | | | Fish (7066) | | | | Coke (14,513) | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Degree of B-spline spline volume | $2\times2\times2$ | | $3\times3\times3$ | | $2\times2\times2$ | | $3\times3\times3$ | | $2\times2\times2$ | | $3\times3\times3$ | | $2\times2\times2$ | | $3\times3\times3$ | |
| Num of sampling points ($P$) | 21 | 105 | 36 | 105 | 21 | 105 | 36 | 105 | 21 | 105 | 36 | 105 | 21 | 105 | 36 | 105 |
| Accurate FFD (ms) | 1 | 2 | 2 | 3 | 2 | 3 | 4 | 6 | 3 | 6 | 8 | 13 | 6 | 14 | 18 | 29 |
| UUS (ms) | 8 | 9 | 9 | 14 | 8 | 13 | 14 | 22 | 9 | 24 | 23 | 39 | 18 | 48 | 45 | 78 |

## Acknowledgment

This work is supported by the National Natural Science Foundation of China under Grant Nos. 60933007 and 61170138, the 973 program of China under Grant No. 2009CB320801, the Program for New Century Excellent Talents in University under Grant No. NCET-10-0728. We would like to thank Dr. Hongwei Lin for his valuable suggestions.

## References

[1] Feng J, Heng PA, Wong TT. Accurate B-spline free-form deformation of polygonal objects. J Graph Tools 1998;3(3):11–27.
[2] Feng J, Peng Q. Accelerating accurate B-spline free-form deformation of polygonal objects. J Graph Tools 2000;5(1):1–8.
[3] Vergeest JSM. Cad surface data exchange using step. Comput Aided Des 1991;23(4):269–81.
[4] Barr AH. Global and local deformations of solid primitives. SIGGRAPH Comput Graph 1984;18(3):21–30.
[5] Sederberg TW, Parry SR. Free-form deformation of solid geometric models. SIGGRAPH Comput Graph 1986;20(4):151–60.
[6] Gain J, Bechmann D. A survey of spatial deformation from a user-centered perspective. ACM Trans Graph 2008;27(4):107:1–21.
[7] Parry SR. Free-form deformation in a constructive solid geometry modeling system. Ph.D. thesis; Brigham Young University; Provo, UT, USA; 1986. UMI order no. GAX86-16254.
[8] Griessmair J, Purgathofer W. Deformation of solids with trivariate B-splines. In: Hopgood F.R.A., Strasser W, editors. Proceedings of eurographics; 1989. p. 137–48.
[9] Gain JE, Dodgson NA. Adaptive refinement and decimation under free-form deformation. In: Eurographics UK 99, Cambridge, vol. 17; 1999. p. 13–5.
[10] Sánchez-Reyes J, Chacón J. Hermite approximation for free-form deformation of curves and surfaces. Comput Aided Des 2012;44(5):445–56.
[11] Feng J, Nishita T, Jin X, Peng Q. B-spline free-form deformation of polygonal object as trimmed bzier surfaces. Visual Comput 2002;18:493–510, http://dx.doi.org/10.1007/s00371-002-0171-1.
[12] Shreiner D, Group TKOAW. OpenGL programming guide: the official guide to learning OpenGL, versions 3.0 and 3.1, 7th ed. Addison-Wesley Professional; 2009, ISBN: 0321552628, 9780321552624.
[13] Chua C, Neumann U. Hardware-accelerated free-form deformation. In: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS workshop on graphics hardware. HWWS '00. New York, NY, USA: ACM; 2000. p. 33–9, ISBN: 1-58113-257-3.
[14] Coquillart S. Extended free-form deformation: a sculpturing tool for 3D geometric modeling. SIGGRAPH Comput Graph 1990;24(4):187–96, http://dx.doi.org/10.1145/97880.97900 URL 〈http://doi.acm.org/10.1145/&97880.97900.
[15] Schein S, Elber G. Real-time free-form deformation using programmable hardwares. Int J Shape Model 2006;12:179–92.
[16] NVIDIA. Chapter 1. Introduction. The Cg tutorial; 2012 〈http://http://http.developer.nvidia.com/CgTutorial/cg_tutorial_chapter01.html.
[17] Hahmann S, Bonneau GP, Barbier S, Elber G, Hagen H. Volume-preserving FFD for programmable graphics hardware. Visual Comput 2012;28:231–45, http://dx.doi.org/10.1007/s00371-011-0608-5.
[18] Modat M, Taylor Z, Barnes J, Hawkes D, Fox N, Ourselin S. Fast free-form deformation using the normalised mutual information gradient and graphics processing units. Med Phys 2008;98(3):278–84.
[19] Modat M, Ridgway GR, Taylor ZA, Lehmann M, Barnes J, Hawkes DJ, et al. Fast free-form deformation using graphics processing units. Comput Methods Prog Biomed 2010;98(3):278–84.
[20] Farin G. Curves and surfaces for CAGD: a practical guide. 5th ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc.; 2002 ISBN: 1-55860-737-4.
[21] NVIDIA. Cuda occupancy calculator. NVIDIA GPU computing documentation; 2012 〈http://developer.download.nvidia.com/compute/DevZone/docs/html/C/tools/CUDA_Occupancy_Calculator.xls〉.
[22] Schwarz M, Stamminger M. Fast GPU-based adaptive tessellation with CUDA. Comput Graph Forum 2009;28(2):365–74 URL 〈http://dblp.uni-trier.de/db/journals/cgf/cgf28.html#SchwarzS09〉.
[23] Concheiro R, Amor M, Bóo M, Doggett MC. Dynamic and adaptive tessellation of bézier surfaces. In: Richard P, Braz J, editors. GRAPP. SciTe Press; 2011. p. 100–5 ISBN: 978-989-8425-45-4.
[24] Rost RJ, Licea-Kane B, Ginsburg D, Kessenich JM, Lichtenbelt B, Malan H, et al. OpenGL shading language. 3rd ed.Addison-Wesley Professional; 2009 ISBN: 0321637631, 9780321637635.
[25] Hu SM, Zhang H, Tai CL, Sun JG. Direct manipulation of FFD: efficient explicit solutions and decomposible multiple point constraints. Visual Comput 2001;17(6):370–9.