Improved Smooth Free-Form Deformation with Sharp Feature Awareness

Zheqi Lu, Jieqing Feng*

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

Abstract

In an smooth free-form deformation [1] of a polygonal object, the linear geometry, e.g., triangles or planar polygons will be subdivided against a knot box, but it will produce pathological triangles which will contribute to many problem about efficiency and robustness. On the other hand, cui utilize the parallel computing power of the GPU via CUDA, which is limited to NVIDIA hardware. Thus, his method can not be used on neithor AMD GPU nor mobile platform. In this paper, we propose an approach which will be used to clip the original model to avoid pathological case. Moreover, we implement the whole framework of smooth FFD with OpenGL compute shader in order to make it more general. Our subdivision method will produce sub-triangles as regular as possible, but cannot guarantee that each sub-triangle lies in a knot box.Accordingly the method will introduce acceptable errors. However, the trade-off is worthwhile in accordance with Error analysis. We also extend the deformation framework to the mobile phone by the aid of the good across-platfor of OpenGL.

Keywords: smooth FFD, sharp feature preserving, GPU, OpenGL compute shader, normal field

1. Introduction

Free-form deformation (FFD) is a prevalent shape manipulation and shape animation method in computer graphics and geometric modeling [2]. Classic FFD is conducted on the sampled points of the geometric model. However, the approach tends to produce an aliased deformation result when using a low sampling density.

Free-form deformation (FFD[2]), as well as many kinds of variants, is useful and widely used whether in academia or in industry when refer to computer graphics and geometric modeling. However, Classic FFD is very simple and efficient but also has many limitation, it is conducted on the sampled points of the geometric model and the approach tends to produce an aliased deformation result when using a low sampling density.

As an alternative, smooth FFD [3, 4, 1] which is on the foundation of accurate FFD (Feng et al., 1998, 2002; Feng and Peng, 2000) deforms the planar polygons as a set of triangular Bzier patches based on the functional composition of Bernstein polynomials (DeRose, 1988; DeRose et al., 1993). The deformation result is smooth and sharp feature awareness. However, In the pre-processing phase, the pathological sub-triangles producted by clip process will worsen efficiency and robustness. So we propose another subdivision approach to get more regular sub-triangles with the cost of a little increase of error. On the other hand, the framework's applicability and generality are bad due to the limit of CUDA. In order to address it, We implement our compute-intensive task with OpenGL compute shader. The main contributions of the paper are summarized as follows:

- A new clip method are proposed to get more regular sub-triangles.
- Implement the framework by OpenGL compute shader for better applicability and generality.

*Corresponding author

Email address: jqfeng@cad.zju.edu.cn (Jieqing Feng)

What's more, graphics applications is move to mobile platform due to the development of mobile hardware. So we also port our work on Android platform. It needs little addition effort thanks to the portable OpenGL.

2. Related Work

FFD, which was first proposed by Sederberg and Parry [2], is an intuitive model manipulation and soft object animation method. The main concept of FFD is to embed the object into an intermediate space, e.g., a Bézier volume. Users first edit the shape of the intermediate space; then, the space deformation is transferred to the embedded object, whereas the topological connectivity of the object remains unchanged. There are many successive studies regarding FFD. Most of these studies focus on improving the interactive means of FFD [5, 6, 7, 8, 9]. Gain and Bechmann [10] provided a detailed survey of these methods.

Traditional FFD and its extensions deform the sampled vertices of the model. Thus, the quality of the deformation result depends on the sampling density of the vertex. As a solution to the sampling problem, adaptive upsampling approaches [11, 12, 13] are more efficient than the naive uniform upsampling approach on CPU. The adaptive upsampling approaches consider the polygon size or surface curvature and upsample the model if necessary. But they cannot handle certain special or pathological cases well, and are difficult to be ported on GPU. Accurate FFD, which was proposed by Feng et al. [14, 15, 16], is an alternative approach to solving the sampling problem. However, it is computationally intensive, and it also consumes considerable bandwidth, i.e., transferring a large amount of data from the CPU to the GPU after the intensive computations are performed in the CPU. Thus, the algorithms are not interactive or performed in real time in practical applications.

In the recent years, GPUs have been widely adopted for FFD implementations due to their tremendous parallel computing power. Chua et al. [17] proposed an OpenGL-oriented hardware evaluator sub-system to accelerate FFD evaluations. However, none of the GPU vendors integrate this type of dedicated sub-system into their GPUs. In contrast, GPUs have evolved into general-purpose many-core processors. Schein et al. [18] implemented a GPU-accelerated FFD using the NVIDIA CG language. Jung et al. [19] achieved the same goal using NVIDIA CUDA and embedded it to the X3D system. Hahmann et al. [20] proposed a GPU-based, volume-preserving FFD. They employed the multilinear property of volume constraint and derived an explicit solution. The GPU acceleration component implemented by CUDA is 6.5-times faster than its CPU counterpart.

Cui and Feng (2013, 2014) proposed GPU-based accurate FFD of polygonal objects, the results of which are represented in terms of trimmed tensor product Bzier patches or triangular Bzier patches. They are sufficiently efficient to meet the real-time or interactive demands of large-scale models. However, the deformation is only performed on the linear geometry, without considering the normal of the model. Smooth FFD (Cui and Feng 2015) is proposed by Cui and Feng (2015), which adjust the deformed the results of above work according to the normal information of the model so that to obtain a smooth free-form deformation with visually plausible smooth geometry and shading. However, Smooth FFD will generate degenerative triangles and narrow triangle which will do damage accuracy and efficiency in many areas such as CAD, PDE-based computation and so on. And when it comes to generality, Smooth FFD is implemented by CUDA which will limit their whole framework to NVIDIA hardware.

On the other hand, there're more and more graphic applications on the mobile platform. Mobile platform will develop so further that it not only show the 3D things, but also has the demand to edit the 3D model.[21], [22] already do some work on mobile platform. We want to port a variant FFD method to mobile phone.

Therefore, we propose a method which is based Smooth FFD which can run on multi-platform including mobile platform to address the above problems.

3. Overview of Accurate FFD in Terms of Triangular Bézier Patches

The GPU-based accurate FFDs [3, 4] of polygonal objects adopt trimmed tensor product Bézier patches and triangular Bézier patches as the deformation result, respectively. In this paper, we adopt the framework

of accurate FFD using triangular Bézier patches [14, 16, 4] since it is more efficient than the one using trimmed tensor product Bézier patches. Some basic notation is introduced below.

$\mathbf{R}(u, v, w)$ is a B-spline volume of degree $n_u \times n_v \times n_w$ with $m_u \times m_v \times m_w$ control points:

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

where $\{\mathbf{R}_{ijk}\}_{i=0,\ j=0,\ k=0}^{m_u-1,m_v-1,m_w-1}$ are the control points, $\{N_{i,n_u}(u)\}_{i=0}^{m_u-1}$, $\{N_{j,n_v}(v)\}_{j=0}^{m_v-1}$ and $\{N_{k,n_w}(w)\}_{k=0}^{m_w-1}$ are normalized B-spline basis functions, and $\{u_i\}_{i=0}^{n_u+m_u}$, $\{v_i\}_{j=0}^{n_v+m_v}$ and $\{w_k\}_{k=0}^{n_k+m_k}$ are the knot vectors along the u, v and w directions, respectively. Every three-dimensional 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 [14, 16, 4], each polygon of the model is first clipped against the knot boxes such that the generated sub-polygons lie inside of a knot box. Second, the generated sub-polygons are triangulated. The accurate FFD of such a sub-triangle in a knot box governed by $\mathbf{R}(u, v, w)$ is a triangular Bézier patch [14, 16], whose degree is $n = n_u + n_v + n_w$. Let the triangular Bézier patch be denoted as $\mathbf{P}(u, v, w)$:

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

where $\{B^n_{ijk}(u, v, w) = \frac{n!}{i!j!k!} u^i v^j w^k \mid i + j + k = n\}$ are the Bernstein basis functions defined on a 2D simplex, *i.e.*, a triangle. Its control points are $\{\mathbf{P}_{ijk} \mid i + j + k = n\}$, which can be efficiently computed via polynomial interpolations [16].

In Smooth FFD, each triangle and its normal field are deformed as two cubic triangular Bzier patches which are generated via constrained fitting method whose input are the sample points sampled from B-Spline body. Then, the curved geometry corresponding to the deformed triangles is locally adjusted to tone the smoothness of the geometry appearance according to the deformed normal field. As a result, a smooth free-form deformation with visually plausible smooth geometry and shading is obtained.

4. Clip Triangle

The degenerate triangles and skinny triangles generated by clipping against knot cage will have many problem in numerical precision, cross-product for normal, CFD solution and so on. According to [14, 16], Accurate FFD as well as Smooth FFD will clip the input model against the knot boxes in order to get accurate result of FFD. However, the final result of Smooth FFD is describe as cubic triangular Bzier patch which is only a approximation rather than the accurate high-order one. It is a success trade-off between precision and efficiency. Therefore, the limitation that input model should be clipped against the knot boxes can be remove for the result already not accurate.

Thus, the weakness introduced by bad quality triangle(see 1) can also be conquered if we ask for a method which can clip model to triangles as regular as possible rather than clip against knot boxes. On the other side, for two triangles which have the common edge, the clipping scheme generated by our clipping method should be the same on the common edge to avoid cracks.

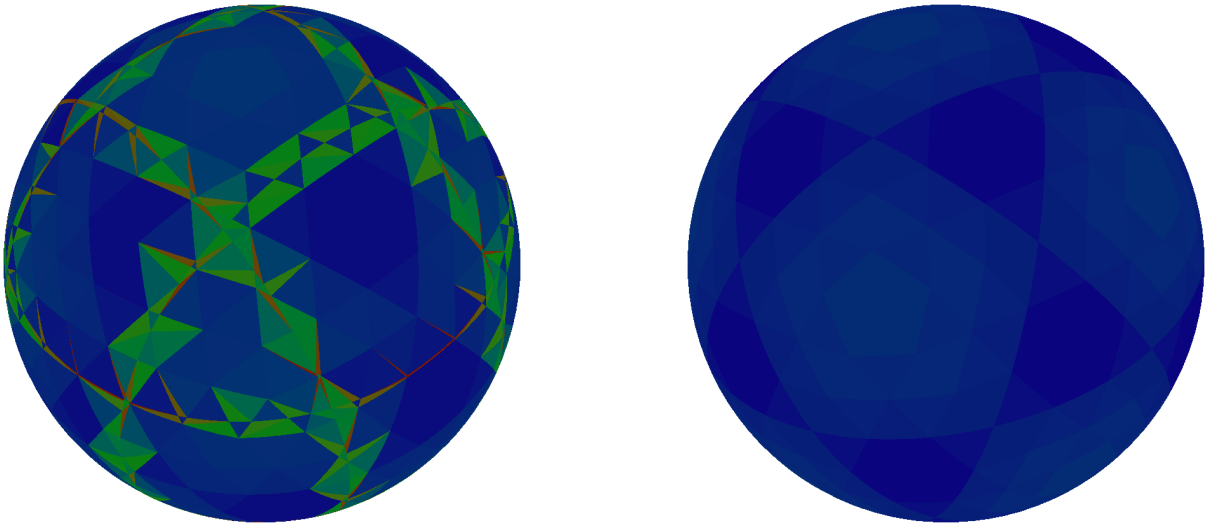
So our clipping algorithm should archive two goal as follows:

- the length of all sub-triangles side should be as equal as possible. Thus, the quality of the sub-triangles is guarantees and the area of sub-triangle will be as similarly to each other as possible.
- the adjacent triangles should have the same clip scheme on the common edge to avoid cracks.

4.1. The overview of the clipping algorithm

In order to satisfy the first goal, we introduce a parameter l as out algorithm's input. All the sub-triangles' edge length should as close to l as possible.

First, we defines some symbols for describing our clip algorithm. e_i is the edges of the Triangle t with $i = 0, 1, 2$. le_i is the length of each edge. Every edge will be clip in to $round(le_i/l)$ segments where l is a



(a) the clipped model's triangle quality

(b) the original model's triangle quality

Figure 1: clipping demonstrate

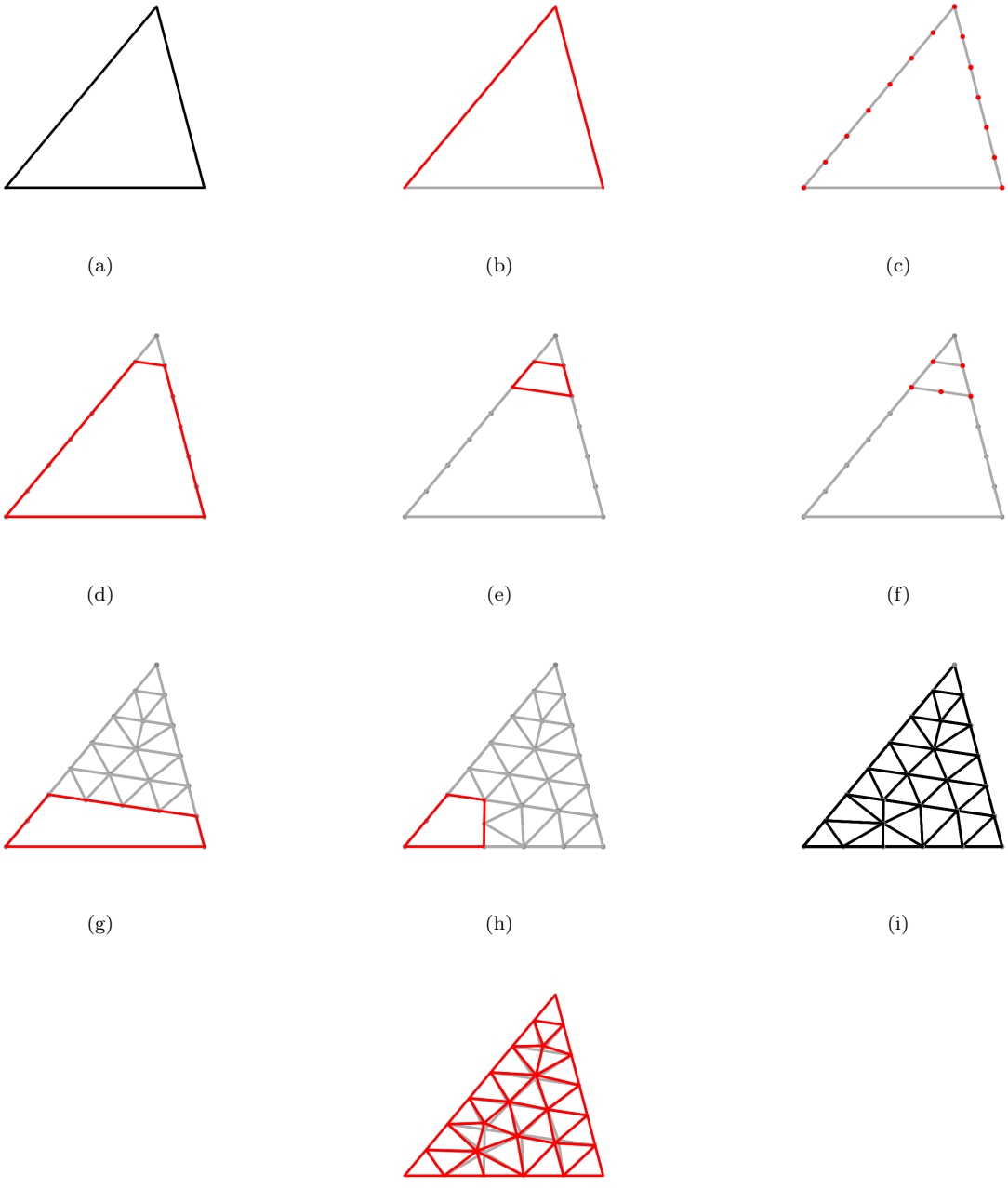
constant parameter which can control the size of the sub-triangles and round and length are the round and length function. p_{ij} is the jth point of the e_i generated by previous clip process.

1. Find the minimize interior angle of the triangle. We can assume that two sides of the angle is e_0, e_1.
2. Compute the clipping points p_{ij} on e_0 and e_1 which divide corresponding edge into $round(le_i/l)$ segments evenly. n and m is the number of the clipping points of e_0 and e_1. The indices of clipping points on the two edge are increase from 0 to n and m along the direction from the vertex whose angle is minimize to the other vertex on the edge. Next, we cut-off the first sub-triangle $e_{00}e_{01}e_{11}$.
3. The remain part of the triangle is a quadrilateral. We cut off $e_{0i}e_{1i}e_{1(i+1)}e_{0(i+1)}$, segment its edge like step 2 and finally clipping it to sub-triangles. The procedure is iterative. If $n = m$ it repeats for i change from 1 to $min(n, m) - 1$ then go to step 5, otherwise i change from 1 to $min(n, m) - 2$ and go next step.
4. The remain part is a quadrilateral which is similar to the initial state of step, we can recursively call the process of step 3 until whole input triangle are clipped.
5. Use CVT method iteratively for 5 times to optimize the position of inner vertexes of the sub-triangles to get more regular sub-triangles.

What not mentioned above is some corner case at the end of recursion described in step 3. The last remain part of the triangle maybe can't be handled with step 3. However there's only several corner cases, and we can provide clipping scheme for every of them.

4.2. the number of iterations

According to 3, we can see that the improvement of CVT is little after the third time. So we iterative cvt for 5 times.



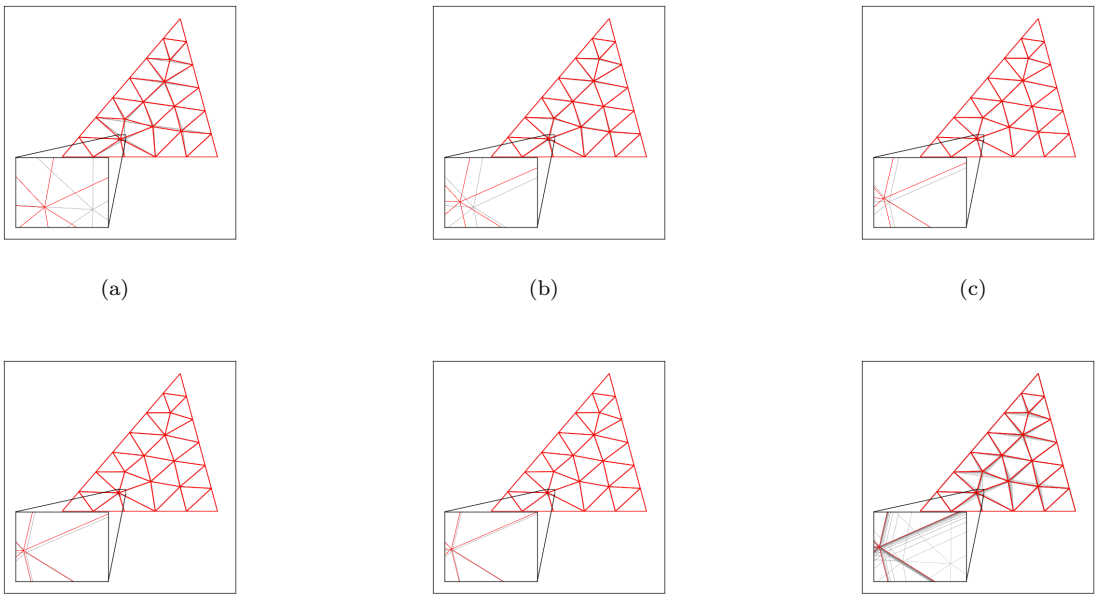
(a) (b) (c)

(d) (e) (f)

(g) (h) (i)

(j)

Figure 2: clipping demonstrate



(a) (b) (c)

(d) (e) (f)

Figure 3: The red triangles is the result current cvt, The gray is the previous result of cvt

4.3. Some discuss of l

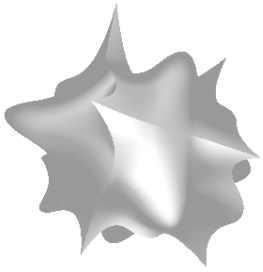
Our new algorithm will receive a parameter l as the approximate maximum length of the sub-triangles' sides as description above. It is an essential parameter which will effect the number of sub-triangle as well as the error between our final result and Accurate FFD's result. We do a lot of work to find out appropriate l which can minimize the error of the final result for all kinds models from the simple ones formed by a little patches to the fine ones composed by a large number of tiny patches. The result is show in 4, we can find that these is no general l for all kinds of models. However, it can see that the error is had obviously positive correlation with sub-triangle number. What's more, we cannot tell the results of accurate FFD and our method when the l is smaller than the knot cage edge length. So l can be turned from as small as possible to cage size by users according to their demand of accuracy.

4.4. Implement in GPU

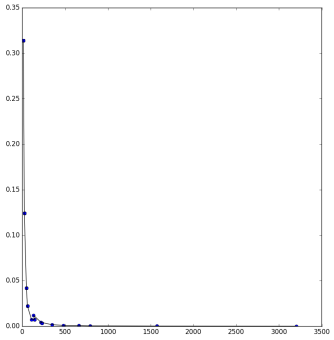
The above algorithm has good result, while it is flexible for different accuracy, but it's to complex to be implemented in OpenGL Compute Shader. Further more, there are a lot of repeated calculation for the model formed by many triangles which have the same size. Hence, we first compute all possible cases of the clipping of triangles, and restore the result in a table. When we really clip a triangle, we can get a pattern from the table according the size of the triangle and the factor l. We trade space for time and the cost of space is reasonable. Of Course, Our GPU clipping triangle algorithm will be faster than the Smooth FFD. The comparison of two algorithm can be see in 1.

5. OpenGL compute shader

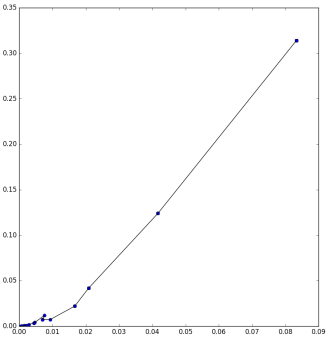
CUDA which is used for implement Smooth FFD is a parallel computing platform and programming model invented by NVIDIA. It enables dramatic increases in computing performance by harnessing the power of the graphics processing unit (GPU). Nevertheless, only on the NVIDIA GPU can CUDA runs. It lead to



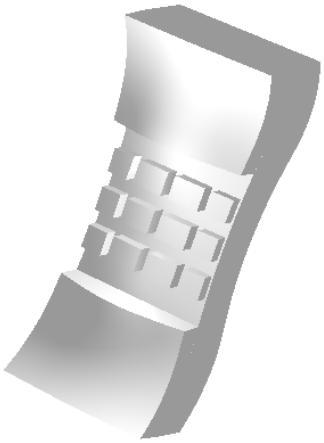
(a)



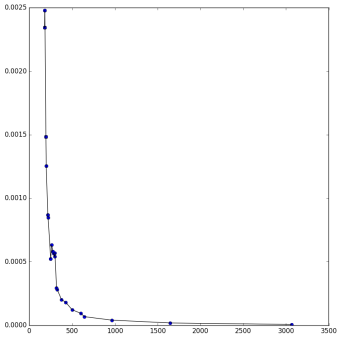
(b)



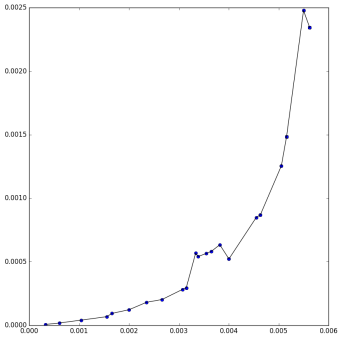
(c)



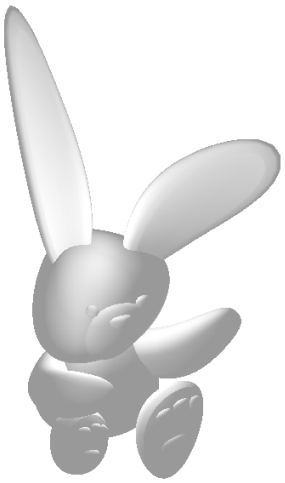
(d)



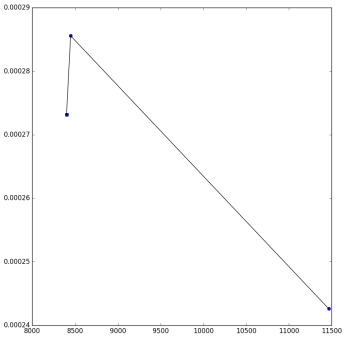
(e)



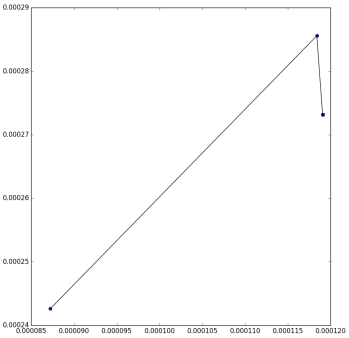
(f)



(g)



(h)



(i)

Figure 4: Relationship of l and sub-triangles number

the compatibility problems in other desktop GPU as well as mobile platform which is rapid developing in recent years.

While Compute Shader is an OpenGL Shader Stage that is used entirely for computing arbitrary information. It is available in every platform as long as it supports OpenGL 4.3.

For this reason, we implement our improved method with OpenGL Compute Shader.

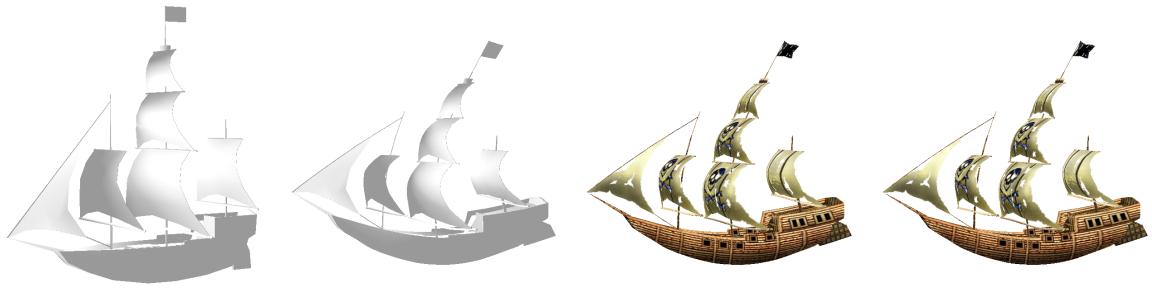
6. Implementation Results and Comparison

The proposed method is implemented on a PC with an Intel Core i7 4710MQ CPU@2.50GHz, 8 GB of main memory and an NVIDIA GeForce GT 730M GPU. The operating system is Arch Linux x86_64. The CPU and GPU components of our method are written using python and OpenGL compute shader, respectively.

We will compare our proposed improved smooth FFD with [1] from the aspects of rendering result, efficiency and approximation errors.

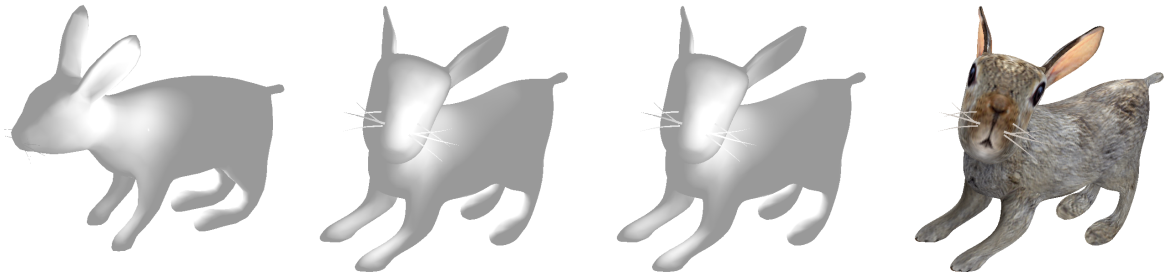
6.1. Comparison of the rendering results

The deformation results are shown in Figure 5 and 6. Each triangular Bézier patch is tessellated into 100 triangles. There are 4 sub-figures in each of the 2 examples: (a) is the original model; (b) is the result of smooth FFD [3, 4]; (c) is the result of improved smooth FFD; (d) is the textured shading effect of (c).



(a) (b) (c) (d)

Figure 5: Deformation of the Ship model by a $3 \times 3 \times 3$ B-spline volume with $5 \times 8 \times 5$ control points



(a) (b) (c) (d)

Figure 6: Deformation of the Rabbit model by a $2 \times 2 \times 2$ B-spline volume with $5 \times 8 \times 5$ control points

These figures illustrate that the results obtained with smooth FFD and improved smooth FFD are almost the same. We can not tell each other with eyes. The highlight is smooth because the normals that we use

(a) rendering result of Cui et al.[1]

(b) rendering result of our mothod

Figure 7: Model used by comparison of efficiencies

are independent of the model geometry. The smooth geometry and sharp features benefit from our geometry and normal adjustment method.

6.2. Comparison of efficiencies

The efficiencies obtained with improve smooth FFD and [1] are compared in Table 1 and Table 2. The model we use is shown in Figure 7. This model has 46,742 faces. As in Section 6.1, each triangular Bézier patch is tessellated into 100 sub-triangles. The degree of the B-spline volume is $2 \times 2 \times 2$, with $5 \times 5 \times 5$ control points. Table1 illustrates that the speed of our method is faster than that of [1] due to clip triangle by GPU which [1] do it by CPU. In the deformation stage, the total time of all stages of Cui's method is 65.464ms. Our method just has an overall run time, 67.419ms, because we implement all of the stages in the same shader while there's not a suitable method to measure time inside a shader. the two method have almost the same efficiencies. The proposed method is sufficiently fast to handle large models in real time.

Table 1: Comparison of the efficiencies of our method and [1](pre-compute stage)

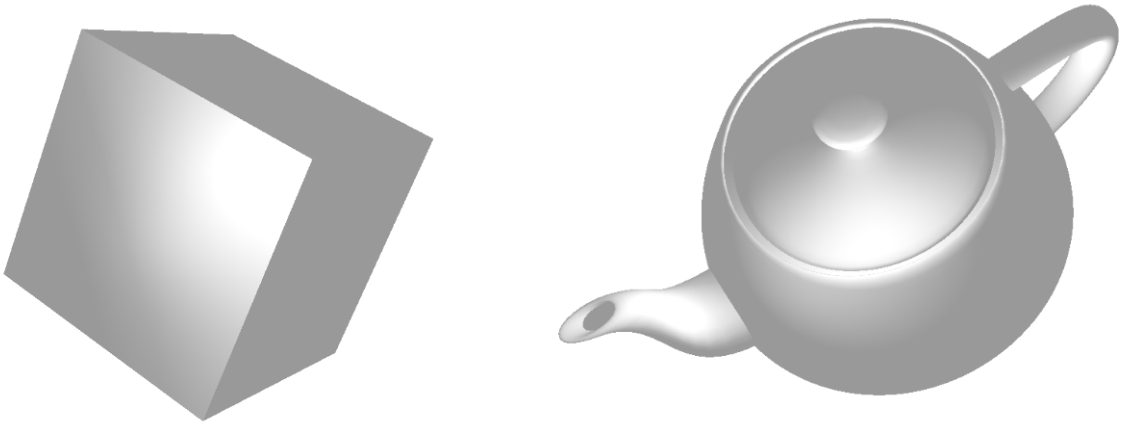
Step	Our method	[1]
Generate PN-triangle	4.618	104.465
Clip triangle	21.475	5027.418
total	26.093	5131.883

Table 2: Comparison of the efficiencies of our method and [1](deform stage)

Step	Our method	[1]
Calculate the sampling points	67.419	30.522
Calculate the constraint points		10.874
Calculate the control points		11.125
Calculate the adjusting normals		4.209
Adjust the control points		4.344
Calculate the tessellation points		4.057
total	67.419	65.464

6.3. Approximation error tests

In the improved smooth FFD as well as smooth FFD, both the geometry and normal are obtained approximately via constrained fitting. Thus, there are approximation errors in the geometry and normal compared with the accurate FFD. Since the smooth parts of a polygonal object can be regarded as a piecewise linear approximation of a potentially smooth shape, which is unknown in general, thus it is difficult to evaluate the approximation errors.

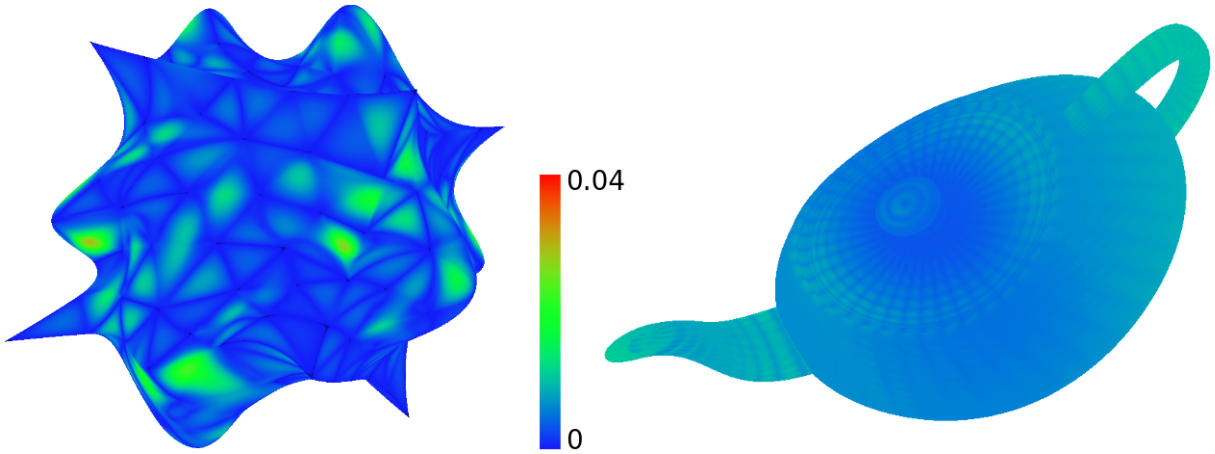


(a) Original model

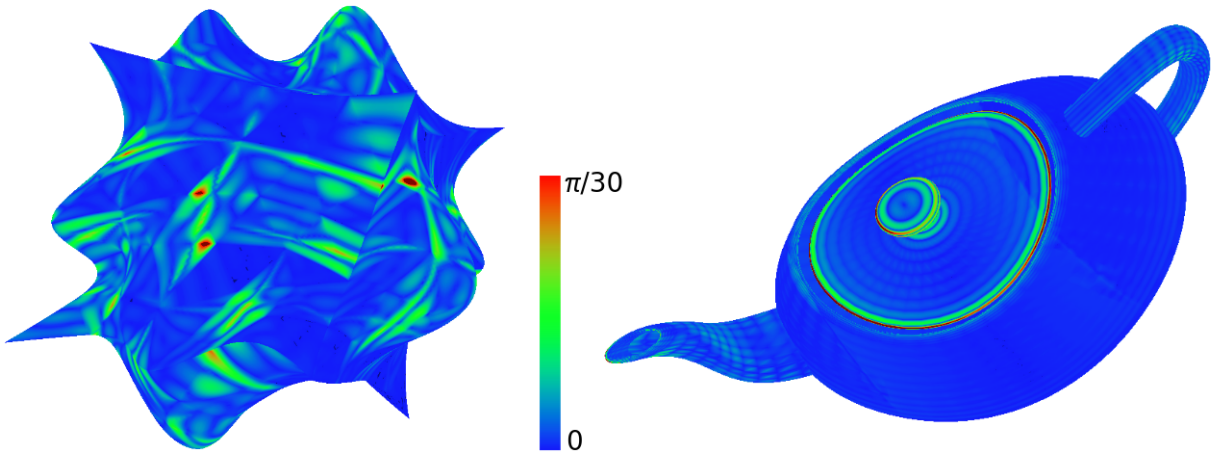
(b) Improved Smooth FFD results of (a)

(c) Smooth FFD results of (b)

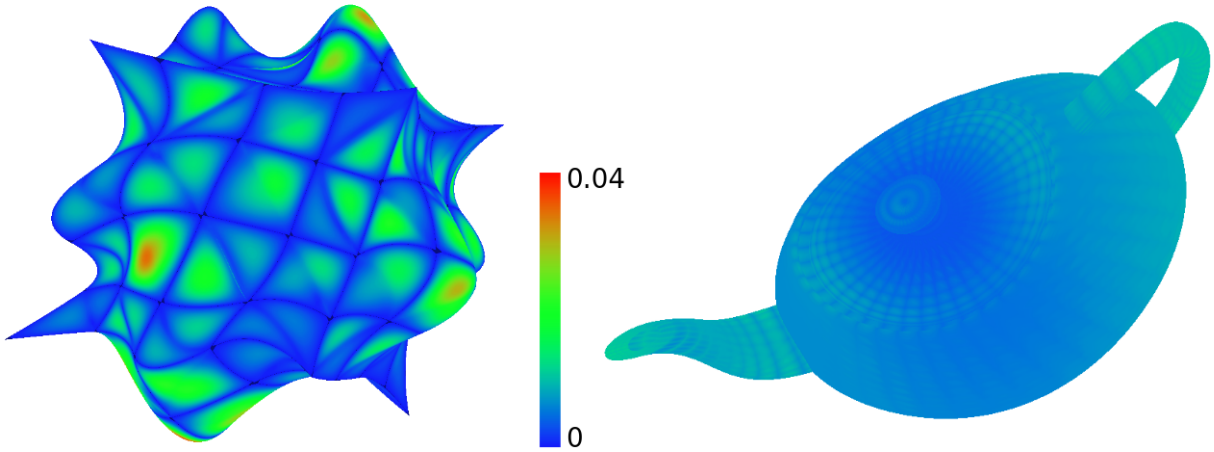
(d) Accurate FFD results of (c)



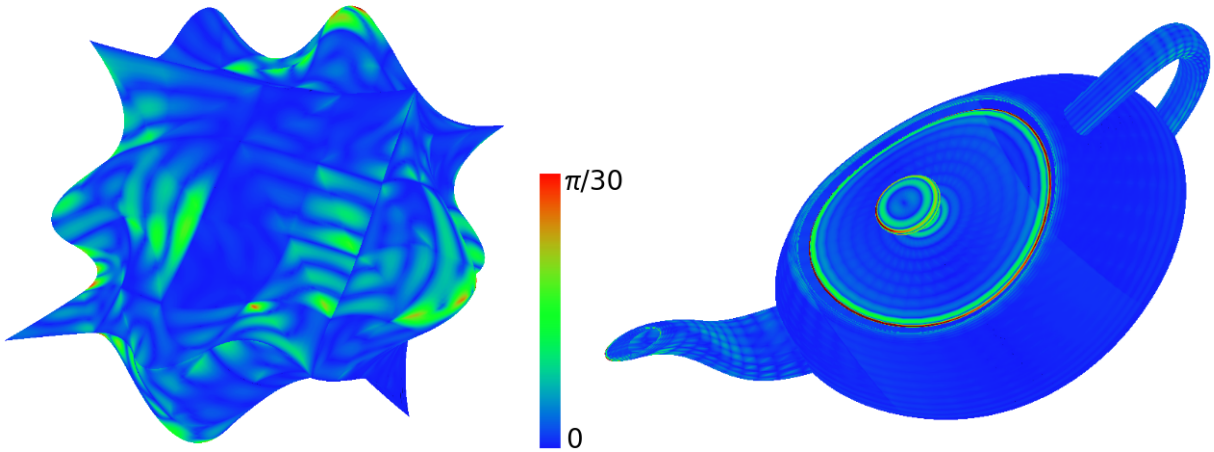
(e) Improved Smooth FFD's Geometry errors



(f) Improved Smooth FFD's Normal errors



(g) Smooth FFD's Geometry errors



(h) Smooth FFD's Normal errors

Figure 6: Error testing

Here, we use a Cube model and a Utah teapot model consists of 36 bicubic Bézier patches to test the approximation error, because the geometry and normal of both the input objects are defined accurately. Both of the two models are normalized into $[-1, 1]^3$. The deformations of the Cube model by [3] and [4] are accurate for both the geometry and normal. The accurate deformation result of Utah teapot is obtained via original FFD of uniformly sampled points and normals. The input of smooth FFD of the Utah teapot is a mesh generated via de Casteljau subdivisions. The error test results are shown in Figure6, where the first column are the Cube model deformed by a $2 \times 2 \times 2$ B-spline volume, respectively. The second column is the Utah teapot deformed by a $2 \times 2 \times 2$ B-spline volume.

The statistics of geometry error, normal error and volume error are given in Tables 5-**??**, respectively. The geometry error is the Euclidean distances between the corresponding vertices, and the normal error is the angles between the corresponding normals.

Table 3: Cube Geometry approximation errors

	Average error	Maximum error
Improved Smooth FFD	0.002952310	0.02908119
Smooth FFD	0.004904391	0.03718415

Table 4: Cube Normal approximation errors

	Average error	Maximum error
Improved Smooth FFD	0.5785680°	15.89719°
Smooth FFD	0.5853341°	21.24638°

Table 5: Teapot Geometry approximation errors

	Average error	Maximum error
Improved Smooth FFD	0.006597950	0.01203453
Smooth FFD	0.006650957	0.01220984

Table 6: Teapot Normal approximation errors

	Average error	Maximum error
Improved Smooth FFD	0.6152188°	22.53011°
Smooth FFD	0.5491496°	22.53010°

According to Figure 6 and Table 5, the proposed improved smooth FFD can generate good approximations of accurate FFD from the geometry, normal. Among them, only maximum normal error of the Utah teapot model is a little bit large. The maximum errors only occur at the spout end and lip which are high curvature parts. Here the input mesh of the Utah teapot is obtained via uniform sampling approach, thus it is not a good polygonal approximation to its original smooth model. It leads to the maximum normal approximation error as above. That is to say the maximum error comes from the polygonal mesh approximation to the smooth object, which is out of scope of the paper.

7. Conclusion and Future Work

In this paper, we proposed a GPU-based smooth FFD with sharp features awareness that addresses the unsmoothness of the normal field and the geometry artifact problems in the framework of accurate FFD. The algorithm can produce a high-quality deformation result. It is a highly parallellizable GPU algorithm and is able to deform a relatively large-scale model in real time. The algorithm is intuitive and can be implemented easily. It can handle relatively coarse meshes and generate smooth deformation results.

The approach can still be improved in several aspects. First, the uniform tessellation of the cubic triangular Bézier patches will generate many unnecessary small triangles. An efficient adaptive tessellation algorithm via GPGPU is an alternative to our method. Second, the approximation error of the smooth FFD for polygonal object is worth to be analyzed in theory. A feasible error bound is useful to guide the discretization of an smooth object, which is essential for generating high-quality deformation result.

8. Acknowledgement

The authors would like to thank the anonymous reviewers who gave valuable suggestions to improve the quality of the paper. This work was supported by the National Natural Science Foundation of China under Grant Nos. 61170138 and 61472349.

References

[1] Y. Cui, J. Feng, Gpu-based smooth free-form deformation with sharp feature awareness, Computer Aided Geometric Design 35 (2015) 69–81.

[2] T. W. Sederberg, S. R. Parry, Free-Form Deformation of Solid Geometric Models, SIGGRAPH Comput. Graph. 20 (4) (1986) 151–160.

[3] Y. Cui, J. Feng, Real-time B-spline Free-Form Deformation via GPU acceleration, Computers & Graphics 37 (1-2) (2013) 1–11.

[4] Y. Cui, J. Feng, Real-time accurate Free-Form Deformation in terms of triangular Bézier surfaces, Appl. Math. J. Chinese Univ. 29 (4) (2014) 455–467.

[5] S. Coquillart, Extended Free-Form Deformation: A Sculpturing Tool for 3D Geometric Modeling, SIGGRAPH Comput. Graph. 24 (4) (1990) 187–196. doi:10.1145/97880.97900.
URL http://doi.acm.org/10.1145/97880.97900

[6] K. Hui, Free-form design using axial curve-pairs, Computer-Aided Design 34 (8) (2002) 583 – 595. doi:10.1016/S0010-4485(01)00132-4.
URL http://www.sciencedirect.com/science/article/pii/S0010448501001324

[7] R. MacCracken, K. I. Joy, Free-Form Deformations with Lattices of Arbitrary Topology, in: Proceedings of the 23rd Annual Conference on Computer Graphics and Interactive Techniques, SIGGRAPH '96, ACM, New York, NY, USA, 1996, pp. 181–188. doi:10.1145/237170.237247.
URL http://doi.acm.org/10.1145/237170.237247

[8] K. T. McDonnell, H. Qin, X. Zhao, PB-FFD: A Point-based Technique for Free-Form Deformation, Journal of Graphics Tools 12 (3) (2007) 25–41.
URL http://cvc.cs.sunysb.edu/Publications/2007/TQ07

[9] G. Xu, K.-C. Hui, W.-B. Ge, G.-Z. Wang, Direct manipulation of free-form deformation using curve-pairs, Computer-Aided Design 45 (3) (2013) 605 – 614. doi:10.1016/j.cad.2012.09.004.
URL http://www.sciencedirect.com/science/article/pii/S001044851200187X

[10] J. Gain, D. Bechmann, A Survey of Spatial Deformation from a User-Centered Perspective, ACM Trans. Graph. 27 (4) (2008) 107:1–107:21.

[11] J. E. Gain, N. A. Dodgson, Adaptive Refinement and Decimation under Free-Form Deformation, Eurographics UK '99, Cambridge 17 (1999) 13–15.

[12] J. Griessmair, W. Purgathofer, Deformation of Solids with Trivariate B-Splines, in: W. S. FRA Hopgood (Ed.), Proceedings of Eurographics, 1989, pp. 137–148.

[13] S. R. Parry, Free-Form Deformation in a Constructive Solid Geometry Modeling System, Ph.D. thesis, Brigham Young University, Provo, UT, USA, uMI order no. GAX86-16254 (1986).

[14] J. Feng, P.-A. Heng, T.-T. Wong, Accurate B-spline Free-Form Deformation of Polygonal Objects, J. Graph. Tools 3 (3) (1998) 11–27.

[15] J. Feng, T. Nishita, X. Jin, Q. Peng, B-spline free-form deformation of polygonal object as trimmed Bézier surfaces, The Visual Computer 18 (2002) 493–510, 10.1007/s00371-002-0171-1.

[16] J. Feng, Q. Peng, Accelerating Accurate B-spline Free-Form Deformation of Polygonal Objects, J. Graph. Tools 5 (1) (2000) 1–8.

[17] C. Chua, U. Neumann, Hardware-Accelerated Free-Form Deformation, in: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS workshop on Graphics hardware, HWWS '00, ACM, New York, NY, USA, 2000, pp. 33–39.

[18] S. Schein, G. Elber, Real-time Free-form Deformation using Programmable Hardwares, International Journal of Shape Modeling 12 (2006) 179 – 192.

[19] Y. Jung, H. Graf, J. Behr, A. Kuijper, Mesh Deformations in X3D via CUDA with Freeform Deformation Lattices, in: R. Shumaker (Ed.), Virtual and Mixed Reality - Systems and Applications, Vol. 6774 of Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2011, pp. 343–351.

[20] S. Hahmann, G.-P. Bonneau, S. Barbier, G. Elber, H. Hagen, Volume-preserving FFD for programmable graphics hardware, The Visual Computer 28 (2012) 231–245, 10.1007/s00371-011-0608-5.

[21] I. Rakkolainen, T. Vainio, A 3d city info for mobile users, Computers & Graphics 25 (4) (2001) 619–625.

[22] F. Lamberti, A. Sanna, A streaming-based solution for remote visualization of 3d graphics on mobile devices, IEEE transactions on visualization and computer graphics 13 (2) (2007) 247–260.