Affine Arithmetic-Based B-Spline Surface Intersection with GPU Acceleration

Hongwei Lin, *Member*, *IEEE*, Yang Qin, Hongwei Liao, and Yunyang Xiong

Abstract—Because the B-spline surface intersection is a fundamental operation in geometric design software, it is important to make the surface intersection operation robust and efficient. As is well known, affine arithmetic is robust for calculating the surface intersection because it is able to not only find every branch of the intersection, but also deal with some singular cases, such as surface tangency. However, the classical affine arithmetic is defined only for the globally supported polynomials, and its computation is very time consuming, thus hampering its usefulness in practical applications, especially in geometric design. In this paper, we extend affine arithmetic to calculate the range of recursively and locally defined B-spline basis functions, and we accelerate the affine arithmetic-based surface intersection algorithm by using a GPU. Moreover, we develop efficient methods to thin the strip-shaped intersection regions produced by the affine arithmetic-based intersection algorithm, calculate the intersection points, and further improve their accuracy. The many examples presented in this paper demonstrate the robustness and efficiency of this method.

Index Terms—Surface-surface intersection, affine arithmetic, GPU acceleration, geometric design

1 Introduction

Surface-surface intersection is a fundamental tool in geometric design software, and it is widely used in geometric operations, such as surface trimming, Boolean operations, and so on. However, the robustness of the surface intersection algorithm is heavily influenced by some singular cases [1], for example, surface tangency, surface overlap, and so on. Additionally, the computation of surface intersection is very complex and time-consuming; therefore, the surface intersection is usually performed in the background [2], which is not desirable for interactive operations.

Interval arithmetic (IA) and *affine arithmetic* (AA) are two range analysis tools often employed in reliability computing. Both tools have been introduced in Bézier surface intersection to improve robustness [3], [4] by recursively decomposing the two parameter domains of the two surfaces. In general, for the same interval input, the range estimated by AA is tighter than that by interval arithmetic; so, in this paper, we use AA to calculate the surface intersection. However, there are two main drawbacks for the AA-based surface intersection algorithm. First, the AA-based surface intersection algorithm calculates the surface intersection by space decomposition, which is implemented by recursion. Because of the algorithm's recursive structure and the complex computation of AA, the speed of the AA-based surface intersection algorithm is slow. Second, due to the overestimation property of AA, the generated surface intersection is a thick strip region and not a curve.

• *The authors are with the Department of Mathematics, State Key Laboratory of CAD&CG, Zhejiang University, Hangzhou, Zhejiang Province 310058, China.*

Manuscript received 1 May 2013; revised 15 Aug. 2013; accepted 17 Sept. 2013; published online 30 Sept. 2013.
Recommended for acceptance by L. Kobblet.
For information on obtaining reprints of this article, please send e-mail to: tvcg@computer.org, and reference IEEECS Log Number TVCG-2013-05-0117.
Digital Object Identifier no. 10.1109/TVCG.2013.237.

To overcome the drawbacks of the AA-based surface intersection algorithm, in this paper, we develop a GPU-accelerated B-spline surface intersection algorithm with AA. First, the affine arithmetic is extended to estimate the range of the piecewise B-spline surface. Second, given a prescribed resolution for parametric space decomposition, only several levels of space decomposition are required for determining the strip-shaped intersection regions on the parameter domains of the two B-spline surfaces with the help of a GPU. The main computations in this step are performed in parallel by CUDA-NVIDIA's GPU parallel computing architecture [5], thus greatly improving the computation speed. Finally, we develop efficient methods to thin the strip-shaped intersection regions, extract the intersection points, and improve their accuracy.

The remainder of this paper is organized as follows: In Section 2, we review the related work on interval analysis and surface-surface intersection. Next, we briefly introduce the main operations of AA in Section 3. In Section 4, the GPU-accelerated surface intersection algorithm with AA is developed in detail. Moreover, we devise the intersection curve generation method in Section 5. Section 6 presents some results and discussions. Finally, Section 7 concludes the paper.

2 Related Work

In this section, we will review the related work on interval analysis and surface intersection, respectively.

2.1 Interval Analysis

Originally, *interval arithmetic* was employed to control the error propagation in numerical computation [6]. Due to its reliability in computation, it has been widely applied in computer graphics [7], [8]. However, the classical interval arithmetic is very conservative in estimating the interval bound. *Affine arithmetic* is an improvement to interval arithmetic, with higher efficiency and more accurate bound estimation [9]. Moreover, a modified version of affine arithmetic can be represented in a matrix form, improving the tightness of bounds [10], [11], [12].

The interval analysis technique has also been introduced in geometric design. By replacing the control points with rectangles or cubes, the interval Bézier curve is used to approximate the floating-point error propagation [13]. Due to the interval representation of the Bézier curve and surface, the intersection computation between curve/curve, curve/surface, and surface/surface is more robust [1], [14]. Furthermore, a validated interval scheme for ordinary differential equation system is employed to trace intersection curve segments between rational parametric surfaces, eliminating the phenomenon of straying and looping robustly [15], [16]. Actually, these methods calculate the intersection by tracing, so it is easy to lose some branches of the intersection. Different from these methods, our method computes the intersection by space decomposition. Therefore, it can find all branches of the intersection in the computation domain.

Instead of using the aforementioned interval Bézier curve and surface, Gleicher and Kass [3] employed interval arithmetic to calculate the Bézier surface intersection by refining the parametric domains recursively. Furthermore, affine arithmetic has been used instead of interval arithmetic in Bézier surface intersection computation to improve the computation accuracy and speed [4]. As aforementioned, the speed of the AA-based surface intersection algorithm is still very slow, and the generated surface intersection is actually a thick strip region, not a curve, due to the overestimation property of AA. In this paper, we accelerate the AA-based surface intersection algorithm by GPU and develop an efficient method to thin the strip-shaped intersection region to a curve.

2.2 Surface Intersection

Surface intersection is a classical problem in geometric design, and there is much literature on this topic. Generally speaking, intersection methods can be classified into the following categories. *Tracing methods* generate the surface/surface intersection curve from an initial starting point [17]. However, when using the tracing method, it is easy to lose some branches of the intersection. *Algebraic methods* represent the surface/surface intersection in implicit form by elimination theory and resultants [18], [19], [20], which are efficient only in computing surface intersection with low degree. Although the exact representation of intersection curve can be obtained by function composition [21], its degree is usually too high to be employed in practice. Finally, *decomposition methods* build polygonal approximations for both surfaces and then intersect the corresponding polyhedral surfaces to produce the intersection curve [22]. Usually implemented recursively, the decomposition method is very time-consuming.

Currently, the widely employed method in practical applications is the hybrid method of decomposition and tracing, which first identifies all intersection branches by decomposition, and then traces out the curve for each intersection branch [23], [24]. To reduce the computation complexity of recursive decomposition, Sinha et al. [25] develop a theorem to detect the topology of transversal intersections. Moreover, Briseid et al. [23] make a pioneering work which accelerates the recursive decomposition algorithm for detecting the topology of intersection by GPU. Furthermore, it is updated to use a heterogeneous system including a multicore CPU and a modern programming model for a GPU [24]. Additionally, Alcantara et al. [26] employ the hash constructed on GPU to determine the intersection of two *animated mesh surfaces* in real time.

Furthermore, in [2], GPU is employed to improve the speed of the decomposition method for generating the final surface/surface intersection curve. By constructing two axis-aligned bounding box (AABB) hierarchies to bound the two surfaces, the overlap tests in [2] are performed level by level on the AABB hierarchies, where a complicated GPU stream reduction algorithm should be implemented to determine the overlapped AABBs. Moreover, the intersection points are extracted by calculating the intersection of the two triangles approximating the subregions of the two surfaces, which makes this method [2] unstable in some singular cases, such as in surface tangency. On the contrary, the AA-based surface intersection algorithm developed in this paper is much easier to implement and is more robust than the method in [2].

For more details on surface intersection, please refer to [27], [28].

3 Affine Arithmetic

Affine arithmetic is one of several models for self-validated numerical computation that have been proposed to address the bound overestimation problem of interval arithmetic [29], [30]. AA provides much tighter range estimates than IA due to its ability to take into account correlations between those quantities. As a result, AA has a higher accuracy, especially for many chained computations for which IA undergoes error explosion.

In affine arithmetic, each input or computed quantity is represented in an affine form:

$$\hat{x} = x_0 + x_1\varepsilon_1 + x_2\varepsilon_2 + \cdots + x_n\varepsilon_n = x_0 + \sum_{i=1}^{n} x_i\varepsilon_i, \qquad (1)$$

where the ε_i are denoted as noise symbols whose values are unknown but are assumed to lie in the interval $[-1, 1]$. Each ε_i stands for an independent source of error or uncertainty that contributes to the total uncertainty of the quantity x. The coefficient x_i is the known real number that gives the magnitude and sign of ε_i. The key feature of affine arithmetic is that the same noise symbol ε_i may appear in two or more affine forms, and the sharing of noise symbols indicates some partial dependence between these quantities. Taking such correlations into account improves the range estimates.

If given an ordinary interval $[\underline{x}, \overline{x}]$, representing a quantity x, the corresponding affine form can be expressed as

$$\hat{x} = x_0 + x_1\varepsilon_1, \quad x_0 = (\underline{x} + \overline{x})/2, \quad x_1 = (\overline{x} - \underline{x})/2. \qquad (2)$$

Conversely, an affine form of $\hat{x} = x_0 + x_1\varepsilon_1 + x_2\varepsilon_2 + \cdots + x_n\varepsilon_n$ can be converted into an interval:

$$[\underline{x}, \overline{x}] = [x_0 - \xi, x_0 + \xi], \quad \text{where } \xi = \sum_{i=1}^{n} |x_i|. \qquad (3)$$

Given two affine forms

$$\hat{x} = x_0 + \sum_{i=1}^{n} x_i \varepsilon_i, \quad \hat{y} = y_0 + \sum_{i=1}^{n} y_i \varepsilon_i, \tag{4}$$

the affine operations in AA in which $c \in \mathbf{R}$ are as follows:

$$\begin{aligned} c \times \hat{x} &= c x_0 + c \sum_{i=1}^{n} x_i \varepsilon_i, \\ c \pm \hat{x} &= c \pm x_0 + \sum_{i=1}^{n} x_i \varepsilon_i, \\ \hat{x} \pm \hat{y} &= (x_0 \pm y_0) + \sum_{i=1}^{n} (x_i \pm y_i) \varepsilon_i. \end{aligned} \tag{5}$$

Multiplication of two affine forms $\hat{x} \times \hat{y}$ is one of nonaffine operations that introduce a new error symbol ε_k whose value is still between $[-1, 1]$:

$$\hat{x} \times \hat{y} = x_0 y_0 + \sum_{i=1}^{n} (x_0 y_i + y_0 x_i) \varepsilon_i + uv\varepsilon_k, \tag{6}$$

where

$$u = \sum_{i=1}^{n} |x_i|, \qquad v = \sum_{i=1}^{n} |y_i|.$$

To improve the robustness and efficiency of computation, we employ the rounded AA [31] in our implementation.

4 B-Spline Surface Intersection with AA

Suppose we are given two B-spline surfaces

$$\begin{aligned} \boldsymbol{P}(u, v) &= (x_p(u, v), y_p(u, v), z_p(u, v)), (u, v) \in [u_s, u_e] \times [v_s, v_e], \\ \boldsymbol{Q}(s, t) &= (x_q(s, t), y_q(s, t), z_q(s, t)), (s, t) \in [s_s, s_e] \times [t_s, t_e]. \end{aligned} \tag{7}$$

In this section, we first extend AA to estimate the range of the piecewise B-spline surface. Then, two strip-shaped intersection regions on the two parameter domains of $\boldsymbol{P}(u, v)$ and $\boldsymbol{Q}(s, t)$ are determined with the help of AA, where the main computation is implemented on NVIDIA's parallel computing architecture, i.e., CUDA. Finally, the strip-shaped intersection regions are thinned, and the intersection points are extracted to form the intersection curve of the two B-spline surfaces, $\boldsymbol{P}(u, v)$ and $\boldsymbol{Q}(s, t)$.

4.1 Region Estimation of B-Spline Surface with AA

Conventionally, AA is defined for globally supported polynomials. In this section, we will extend the AA operations to estimate the region of the piecewise B-spline surface with locally supported basis functions.

A B-spline surface is constructed by a control net and two knot vectors [27]. The following equation gives the analytical definition of a B-spline surface:

$$\begin{aligned} \boldsymbol{P}(u, v) &= (x_p(u, v), y_p(u, v), z_p(u, v)) \\ &= \sum_{i=0}^{n} \sum_{j=0}^{m} N_i^p(u) N_j^q(v) \boldsymbol{P}_{ij}, \\ &\quad (u, v) \in [u_s, u_e] \times [v_s, v_e], \end{aligned} \tag{8}$$

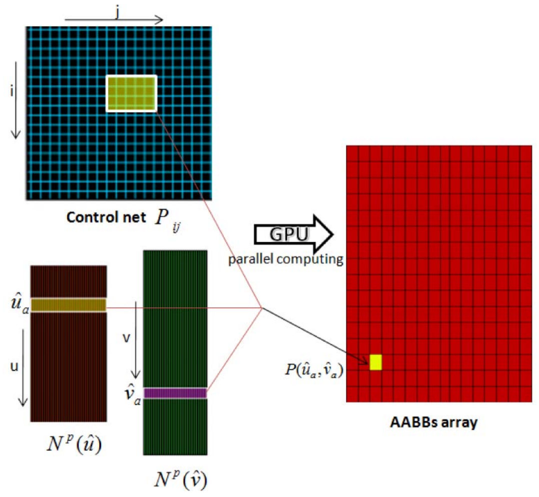


Fig. 1. Affine arithmetic parallel computation by GPU.

where the $\boldsymbol{P}_{ij}$ stand for the control points, and $N_i^p(u)$ and $N_j^q(v)$ are the B-spline basis functions of degree p and q, respectively, which are defined recursively as

$$N_i^p(u) = \frac{u - u_i}{u_{i+p} - u_i} N_i^{p-1}(u) + \frac{u_{i+p+1} - u}{u_{i+p+1} - u_{i+1}} N_{i+1}^{p-1}(u), \tag{9}$$

$$N_i^0 = \begin{cases} 1, & \text{if } u_i \le u < u_{i+1} \\ 0, & \text{otherwise.} \end{cases} \tag{10}$$

The B-spline basis function has the local support property. For some $u \in [u_i, u_{i+1})$, many basis functions of degree p are zero, except for $N_{i-p}^p(u), \ldots, N_i^p(u)$. To calculate a particular point $\boldsymbol{P}(u, v)$ on the B-spline surface, we can ignore the basis functions that are zero at (u, v), just multiply the nonzero basis functions with their corresponding control points, and sum the results.

Given two intervals $[u_a, u_b]$ and $[v_a, v_b]$ for parameters u and v, respectively, we first convert them into the affine forms $\hat{u}$ and $\hat{v}$ (2). For the sake of convenience, we assume that $[u_a, u_b]$ is in the same knot span $[u_i, u_{i+1})$ and similarly for $[v_a, v_b]$ (in $[v_j, v_{j+1})$). Second, the affine arithmetic operations (5), (6) are employed to compute B-spline basis functions. Specifically, we put $\hat{u}$ and $\hat{v}$ into (9) instead of u and v, perform recursive computation via AA operations, and obtain $N_i^p(\hat{u})$ and $N_j^q(\hat{v})$, which are also presented by affine forms and stand for the range estimates of the basis functions. In light of the local support property, we just need the results of $N_{i-p}^p(\hat{u}), \ldots, N_i^p(\hat{u})$ and $N_{j-q}^q(\hat{v}), \ldots, N_j^q(\hat{v})$. Finally, we compute the affine form $\boldsymbol{P}(\hat{u}, \hat{v})$ using the nonzero basis functions and their corresponding control points by (8) (see Fig. 1). $\boldsymbol{P}(\hat{u}, \hat{v})$ means a polygonal approximation to $\boldsymbol{P}(u, v)$, consisting of three intervals, i.e., $x_p(\hat{u}, \hat{v})$, $y_p(\hat{u}, \hat{v})$, and $z_p(\hat{u}, \hat{v})$, each representing a coordinate axis. These intervals construct an AABB in 3D space (see Fig. 2).

If the interval $[u_a, u_b]$ or $[v_a, v_b]$ covers several knot spans, it should be divided into several subintervals, each in a single knot span. Additionally, we construct the AABB for each subinterval by the aforementioned method.

To sum up, if the parametric domain of a B-spline surface is divided into $m \times n$ rectangles along the u and v directions,

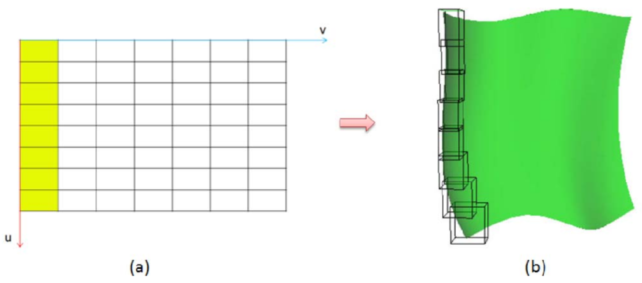


Fig. 2. Decomposition of the parameter space and their corresponding AABBs bounding the B-spline surface. (a) Parameter space, (b) AABBs in 3D space corresponding to the highlighted rectangles in (a).

respectively, we can calculate a set of AABBs using affine arithmetic, in parallel by the GPU (see Fig. 1), which bound the given B-spline surface. Fig. 2 shows the rectangles in the parametric domain (see Fig. 2a) and the AABBs (see Fig. 2b) corresponding to the highlighted rectangles.

4.2 Intersection Region Extraction

To extract the strip-shaped intersection regions on the two parameter domains of the two given B-spline surfaces (7), the two parameter domains should be combined to form a four-dimensional space (u, v, s, t), defined on

$$S^0 = [u_s, u_e] \times [v_s, v_e] \times [s_s, s_e] \times [t_s, t_e]. \tag{11}$$

The conventional AA-based surface intersection algorithm [4] first puts S^0 into the following *overlap test equations*:

$$\begin{cases} R_x &= x_p(\hat{u}_i, \hat{v}_j) - x_q(\hat{s}_k, \hat{t}_l), \\ R_y &= y_p(\hat{u}_i, \hat{v}_j) - y_q(\hat{s}_k, \hat{t}_l), \\ R_z &= z_p(\hat{u}_i, \hat{v}_j) - z_q(\hat{s}_k, \hat{t}_l), \end{cases} \tag{12}$$

and computes the three ranges R_x, R_y, R_z. If the three intervals R_x, R_y, and R_z all contain zero, the pair of AABBs bounding the two surfaces $\boldsymbol{P}(u, v)$ and $\boldsymbol{Q}(s, t)$ overlap each other. Then, we decompose S^0 uniformly into 16 subregions and substitute each subregion into the overlap test equations (12). Otherwise, the parts of the two surfaces bounded by the current pair of AABBs have no intersection, and the subregions corresponding to the current pair of AABBs do not need further handling. This process is performed level by level until the bottom level is reached. Specifically, if the resolution of the bottom level is $2^k \times 2^k \times 2^k \times 2^k$, the conventional method [4] needs a k level computation to reach the bottom level. This procedure costs lots of time.

In this paper, with the help of the GPU parallel computing architecture, we improve the conventional method. The improved GPU-accelerated method can reach the bottom level with resolution $2^k \times 2^k \times 2^k \times 2^k$ only by at most $\lceil \frac{k}{d} \rceil$ level computation, where $\lceil \frac{k}{d} \rceil$ denotes the smallest integer not less than $\frac{k}{d}$, and $2^d \times 2^d \times 2^d \times 2^d$ is the largest GPU parallel thread number. In our implementation, we set $d = 4$, because the largest parallel thread number of the GPU we employed is $2^4 \times 2^4 \times 2^4 \times 2^4$. The GPU-accelerated method is listed in Algorithm 1 and explained in the following.

Algorithm 1. GPU-accelerated intersection region extraction with AA.

```
   // Suppose the user prescribed
      resolution is $n_1 \times m_1 \times n_2 \times m_2$
1  $N_1 = \lceil \ln n_1 \rceil,\ M_1 = \lceil \ln m_1 \rceil,$
   $N_2 = \lceil \ln n_2 \rceil,\ M_2 = \lceil \ln m_2 \rceil$ ;
2  $k_1 = (N_1 \mod d),\ l_1 = (M_1 \mod d),$
   $k_2 = (N_2 \mod d),\ l_2 = (M_2 \mod d)$ ;
3  If some of $k_1, k_2, l_1$ and $l_2$ equal zero, set them as $d$ ;
4  $N_1 = N_1 - k_1,\ M_1 = M_1 - l_1,$
   $N_2 = N_2 - k_2,\ M_2 = M_2 - l_2$ ;
5  Decompose $S^0$ into $2^{k_1} \times 2^{l_1} \times 2^{k_2} \times 2^{l_2}$ sub-regions,
   and determine the overlapped regions $\{R_{ol}\}$ (in CPU) ;
6  **while** *not reach user prescribed resolution* **do**
7      $k_1 = \min(d, N_1),\ l_1 = \min(d, M_1),$
       $k_2 = \min(d, N_2),\ l_2 = \min(d, M_2)$ ;
8      $N_1 = N_1 - k_1,\ M_1 = M_1 - l_1,$
       $N_2 = N_2 - k_2,\ M_2 = M_2 - l_2$ ;
9      Transfer the overlapped regions $\{R_{ol}\}$ into GPU ;
10     **for** *each overlapped region* $R_{ol}$ **do**
11         Decompose $R_{ol}$ into $2^{k_1} \times 2^{l_1} \times 2^{k_2} \times 2^{l_2}$
           sub-regions (in GPU);
12         Perform the overlap test using Eq. (12) for all
           sub-regions and mark the overlapped sub-regions
           in parallel (in GPU);
13     **end**
14     Transfer the mark list to CPU and extract the
       overlapped regions $\{R_{ol}\}$ in CPU ;
15 **end**
```

Supposing the user prescribed resolution is $n_1 \times m_1 \times n_2 \times m_2$, and the largest parallel thread number of the employed GPU is $2^d \times 2^d \times 2^d \times 2^d$, we first compute

$$N_1 = \lceil \ln n_1 \rceil,\ M_1 = \lceil \ln m_1 \rceil, N_2 = \lceil \ln n_2 \rceil,\ M_2 = \lceil \ln m_2 \rceil,$$

and

$$\begin{aligned} k_1 &= (N_1 \bmod d),\ l_1 = (M_1 \bmod d), \\ k_2 &= (N_2 \bmod d),\ l_2 = (M_2 \bmod d), \end{aligned}$$

where $(N_1 \bmod d)$ is the modulus of N_1 and d. If there are some of k_1, l_1, k_2, and l_2 equal to zero, we set them as d. Next, we decompose S^0 into $2^{k_1} \times 2^{l_1} \times 2^{k_2} \times 2^{l_2}$ subregions, and determine the overlapped regions $\{R_{ol}\}$ using (12). Since the computation amount of this step is relatively small, it is performed in CPU.

Furthermore, if the resolution of the current overlapped regions does not reach the user prescribed one, the overlapped regions $\{R_{ol}\}$ are transferred to GPU. Each of them is decomposed into

$$2^{\min(d,N_1)} \times 2^{\min(d,M_1)} \times 2^{\min(d,N_2)} \times 2^{\min(d,M_2)},$$

subregions (see Algorithm 1), and their overlap tests are performed in parallel in GPU. Meanwhile, the overlapped regions are marked in a *mark list*. After the *for-loop* ends, the mark list is transferred into CPU, and the overlapped regions are extracted according to it. The whole procedure terminates when the current overlapped regions reach the prescribed resolution.

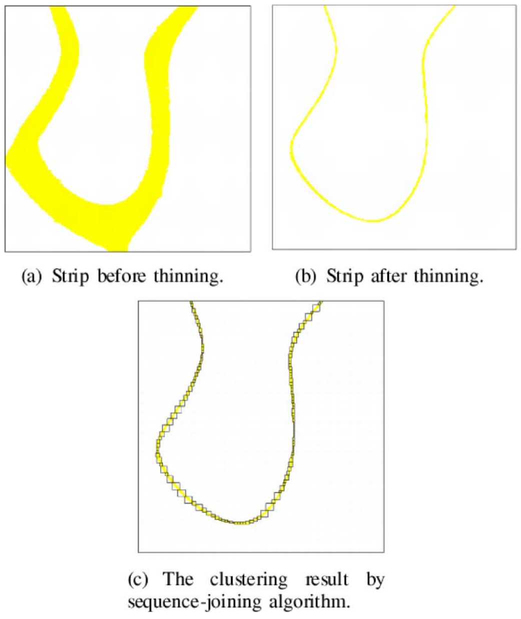

(a) Strip before thinning.
(b) Strip after thinning.
(c) The clustering result by sequence-joining algorithm.

Fig. 3. Strip thinning and clustering by sequence-joining algorithm [32].

Specifically, to transfer data between GPU and CPU, we employ the CUDA library function: *cudaMemcpy(pDeviceMem, pHostMem, size, direction)*. Here, *pDeviceMem* is a pointer to GPU memory, *pHostMem* is a pointer to CPU memory, *size* is the size of the transferred data, and *direction* indicates the data transfer direction. When $direction = cudaMemcpyDeviceToHost$, data are transferred from GPU to CPU; when *direction* = *cudaMemcpyHostToDevice*, data are transferred from CPU to GPU.

5 Intersection Curve Generation

Now, we have obtained two strip-shaped intersection regions in the parameter domains of the two B-spline surfaces $\boldsymbol{P}(u,v)$ and $\boldsymbol{Q}(s,t)$. In fact, they are two sets of rectangles on the two parameter domains, $[u_s, u_e] \times [v_s, v_e]$, and $[s_s, s_e] \times [t_s, t_e]$, corresponding to two sets of AABBs that bound the two surfaces $\boldsymbol{P}(u,v)$ and $\boldsymbol{Q}(s,t)$, respectively. Due to the overestimation of the affine arithmetic, the two sets of rectangles on the two parameter spaces constitute two thick strips, as shown in Fig. 3a. Moreover, the intersection curve will be generated after *strip thinning*, *intersection point generation*, and *accuracy improvement*.

Strip thinning. Due to the overestimation property of the affine arithmetic, the two sets of AABBs bounding the two surfaces, which correspond to the two sets of rectangles on the two parameter domains, are inflated by the affine arithmetic. In fact, the four vertices of a rectangle on a parameter domain correspond to four 3D space points on the B-spline surface. To thin the thick strips on the two parameter domains, we construct the *minimum AABB* bounding the four 3D space points for each rectangle of the thick strips on the parameter domains. Then, we delete the rectangle of each strip whose corresponding minimum AABB does not intersect with any minimum AABB bounding the other surface. In this way, the two thick strips are thinned greatly, as illustrated in Fig. 3b.

Intersection point sequence generation. The thinned strips are also composed of unorganized rectangles. To extract the intersection points, the *sequence-joining method* developed in [32] is employed to cluster the unorganized rectangles. In our implementation, we only perform the sequence-joining algorithm on one strip of intersection region. Without a loss of generality, suppose that the strip is on the parameter space (u,v) of the B-spline surface $\boldsymbol{P}(u,v)$.

First of all, the parameter space (u,v) of $\boldsymbol{P}(u,v)$ is subdivided with resolution same as that of the rectangles in the strip of intersection region. Furthermore, the rectangles composing the intersection strip are taken as *feature rectangles*, and other rectangles as *nonfeature rectangles*. Choosing one feature rectangle as seed, the sequence-joining method [32] constructs one rectangle-like group by continually expanding around the seed level by level, until one of the two following conditions is fulfilled:

1. there is no feature rectangle outside of two opposite boundaries of the current rectangle-like group, or
2. the length of the longest boundary of the current rectangle-like group is greater than the local width of the intersection strip.

After the clustering of this group ends, we choose another nonclustered feature rectangle adjacent to the boundaries of the group as new seed, and a new group can be generated using the method aforementioned. The whole procedure terminates when every feature rectangle is classified into a group. For more details on the sequence-joining algorithm, please refer to [32].

Finally, as the result (see Fig. 3c) of the sequence-joining algorithm, the unorganized feature rectangles are clustered into groups with shapes that are also rectangles with variable sizes. As demonstrated in Fig. 3c, each inner group has two adjacent groups, and the boundary group has one adjacent group. The adjacency relationship between groups naturally assigns an order to these groups, and an ordered intersection point sequence can be generated by them.

Next, we determine one intersection point in each group. As stated above, each group consists of some rectangles. The barycenter of a rectangle corresponds to a point $\boldsymbol{D}_p$ on the B-spline surface $\boldsymbol{P}(u,v)$. However, each rectangle on (u,v) space corresponds to a rectangle on the other parameter space (s,t), the barycenter of which also determines a point $\boldsymbol{D}_q$ on the other B-spline surface $\boldsymbol{Q}(s,t)$. Then, in each group, we retain the rectangle that minimizes the distance between the two points, $\|\boldsymbol{D}_p - \boldsymbol{D}_q\|$, and take $\frac{D_p+D_q}{2}$ as an intersection point.

In this way, the ordered intersection point sequence T_s in 3D space is generated. Moreover, the barycenters of the retained rectangles on the parameter space (u,v) constitute the intersection point sequence I_p on the (u,v) parameter domain, and their corresponding barycenters on (s,t) parameter domain make up the intersection point sequence I_q on the (s,t) domain. In conclusion, we get three pieces of intersection point sequences: one in the 3D space and two on the parameter domains.

Accuracy improvement. Furthermore, the accuracy of the intersection points in T_s can be improved upon. As illustrated in Fig. 4, for each initial intersection point $\boldsymbol{p}_1$, there are two closest points $\boldsymbol{d}_1^1$ and $\boldsymbol{d}_2^1$ on the two surfaces

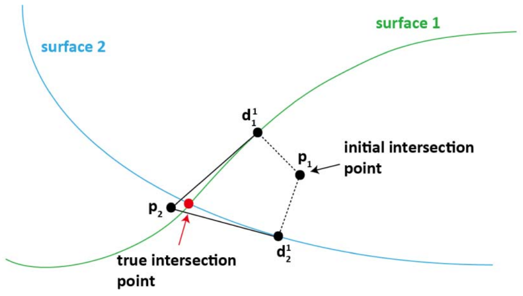


Fig. 4. Sketch of the accuracy improvement using section view of two intersecting surfaces.

$\boldsymbol{P}(u,v)$ and $\boldsymbol{Q}(s,t)$, respectively. As the first improvement step, we replace the original intersection point, i.e., $\boldsymbol{p}_1$ with the point $\boldsymbol{p}_2$, which is the intersection point of three planes, i.e., the two tangent planes of the two surfaces at the two points $\boldsymbol{d}_1^1$ and $\boldsymbol{d}_2^1$, respectively, and the plane determined by the two lines $\boldsymbol{p}_1\boldsymbol{d}_1^1$ and $\boldsymbol{p}_1\boldsymbol{d}_2^1$. The improvement can be performed repeatedly until the intersection point meets the accuracy requirement. In all of our experiments, the accuracy of the intersection points is improved to predefined precision by the above accuracy improvement method. The convergence analysis of the accuracy improvement iterative procedure is presented in the Appendix, which can be found on the Computer Society Digital Library at http://doi.ieeecomputersociety.org/10.1109/TVCG.2013.237.

Finally, we fit the three point sequences I_p, I_q, and T_s, with three pieces of uniform cubic B-spline curves. Two of them are on the two parametric domains, and the other is the intersection curve of the two surfaces: $\boldsymbol{P}(u,v)$ and $\boldsymbol{Q}(s,t)$.

6 Results and Discussions

We implemented the method developed in this paper and tested it on a PC with a 2.66-GHz Intel Core2 Quad CPU Q9400, 3-GB memory, and NVIDIA Geforce GTX 260. For comparison, we modified the CPU-based Bézier surface intersection method using the affine arithmetic developed in [4] to calculate the B-spline surface intersection and tested it on the same platform stated above. Although the method in [4] can generate the thick intersection strips, same as the output of Algorithm 1 developed in this paper (see Fig. 3a), it requires a much longer time than Algorithm 1. Please refer to Table 1 for a comparison. Moreover, to compare our method with the state-of-the-art technique, we implemented the GPU-accelerated B-spline intersection algorithm developed in [2]. The experimental data are listed in Table 2 and will be explained later. Finally, we compared our method with the method in SISL library (SISL method) [33] and listed the experimental data in Table 3.

TABLE 2
Experimental Data of the Method in [2]

Examples	Resolution [a]	time [b]	precision
Fig.5	512^4	0.867	9.7×10^{-4}
Fig.6	512^4	0.912	8.5×10^{-4}
Fig.7	N/A	N/A	N/A
Fig.9	N/A	N/A	N/A
Fig.10	N/A	N/A	N/A
Fig.11	512^4	1.212	9.3×10^{-4}

[a] $512^4 = 512 \times 512 \times 512 \times 512$;
[b] *Time is in second.*

TABLE 3
Experimental Data of the SISL Method [33]

Examples	time [a]	precision	#segments [b]
Fig.5	0.916	4.3×10^{-8}	1
Fig.6	1.155	2.1×10^{-7}	2
Fig.7	136.445	4.9×10^{-2}	107
Fig.9	2.818	6.3×10^{-9}	1
Fig.10	0.062	4.8×10^{-10}	1
Fig.11	169.763	N/A	12

[a] *Time is in second;*
[b] *Number of the intersection segments.*

To measure the precision of the intersection curve, we employ the following *relative root mean square* (RMS) error:

$$R = \frac{\sqrt{\frac{\sum_{i=0}^{n} d_i^2}{n}}}{len}, \quad d_i = d_i(\boldsymbol{P}) + d_i(\boldsymbol{Q}), \tag{13}$$

where $d_i(\boldsymbol{P})$ and $d_i(\boldsymbol{Q})$ are the distances between the ith intersection point and the B-spline surfaces $\boldsymbol{P}(u,v)$ and $\boldsymbol{Q}(s,t)$, respectively, and len is the diagonal length of the bounding box of two intersecting surfaces.

The experimental data for our method are listed in Table 1. In this table, the second column is the bottom-level resolution. Users can specify the bottom-level resolution of the four-dimensional space (u,v,s,t). In the examples illustrated in Figs. 5 and 6, it is set to $512 \times 512 \times 512 \times 512$; in other examples, it is $2,048 \times 2,048 \times 2,048 \times 2,048$.

While the third column of Table 1 is the time cost by our Algorithm 1, i.e., GPU-accelerated intersection region extraction with AA, the fourth column is the time consumed by the CPU-based B-spline surface intersection method using AA, which is modified from the Bézier

TABLE 1
Experimental Data of Our Algorithm and the Method in [4]

Examples	Resolution [a]	GPU time [b]	time of method in [4] [b]	CPU time [b]	precision before	precision after	#iteration
Fig.5	512^4	0.076	23.65	0.62	8.6×10^{-4}	3.8×10^{-11}	3
Fig.6	512^4	0.138	29.30	0.35	5.3×10^{-4}	2.0×10^{-11}	2
Fig.7	2048^4	0.501	205.71	1.32	9.3×10^{-6}	N/A	N/A
Fig.9	2048^4	0.653	248.20	1.93	6.1×10^{-6}	N/A	N/A
Fig.10	2048^4	0.051	16.23	0.007	1.3×10^{-6}	N/A	N/A
Fig.11	2048^4	1.231	465.83[c]	1.73	5.6×10^{-6}	N/A	N/A

[a] $512^4 = 512 \times 512 \times 512 \times 512, 2048^4 = 2048 \times 2048 \times 2048 \times 2048$;
[b] *Time is in second.*
[c] *The algorithm in [4] does not present strategy for computing self-intersection; the time statistics here is cost by the improved method for self-intersection presented in Section 6.*

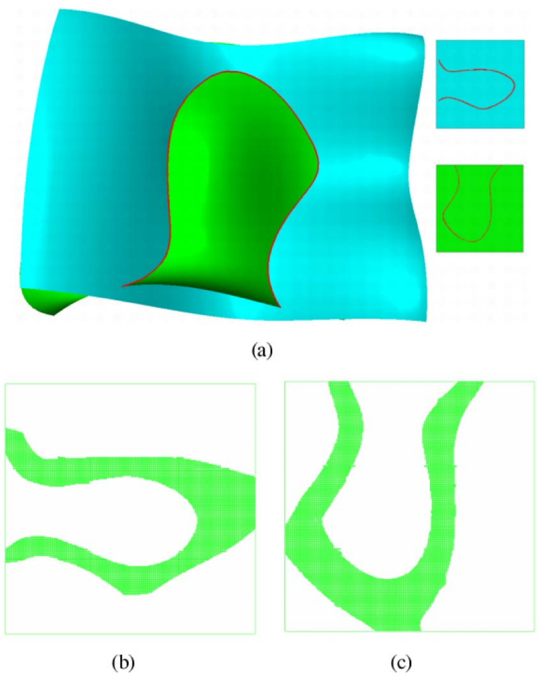


Fig. 5. Two B-spline surfaces intersect with one curve. (a) Result by our algorithm. (b), (c) Result of the algorithm in [4] in the two parametric domains.

surface intersection method developed in [4], with the same output as Algorithm 1. Based on the listed running time, we can see that Algorithm 1 is much faster than the method developed in [4]. The computational speed is raised over two orders of magnitude.

Moreover, the fifth column of Table 1 is the time taken by the algorithm developed in Section 5, including strip thinning, intersection point generation, and accuracy improvement, which is performed in CPU. In addition, the precision for the intersection curve before and after accuracy improvement is recorded in the sixth and seventh columns, respectively. Finally, the last column lists the number of iterations in the accuracy improvement. In the examples illustrated in Figs. 7, 9, 10, and 11, because the bottom-level resolutions are so high that the precisions of the intersections reach a desirable level (i.e., 10^{-6}), we do not perform accuracy improvement for them.

In Table 2, we list the experimental data for the method developed in [2]. In this table, the second column is the bottom-level resolution, the third column is the running time, and the fourth column lists the intersection precision. From Tables 1 and 2, we can see that the running time of our method and the method in [2] is comparable. However, because the method in [2] needs to store a precomputed bounding box hierarchy, and it costs much more memory than our method, the highest bottom-level resolution reached by the method in [2] is $512 \times 512 \times 512 \times 512$. Higher resolution makes the storage of bounding box hierarchy out of memory. At the resolution of $512 \times 512 \times 512 \times 512$, the precision of intersection can attain only 10^{-4} level. Referring to Table 1, our method can reach higher resolution $2,048 \times 2,048 \times 2,048 \times 2,048$, and higher precision level 10^{-6}. Moreover, the method in [2] lacks strategy for handling singular cases, such as tangential intersection (see Figs. 9 and 10), and intersection with two intersecting curves (see Fig. 7), so the data for examples of Figs. 7, 9, and 10 are not available. However, our method is more robust and able to deal with these singular cases successfully.

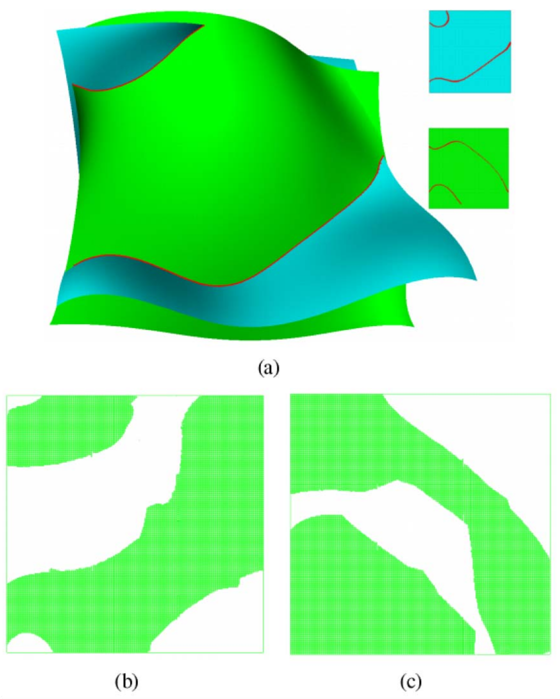


Fig. 6. Two B-spline surfaces intersect with two separate curves. (a) Result by our algorithm. (b), (c) Result of the algorithm in [4] in the two parametric domains.

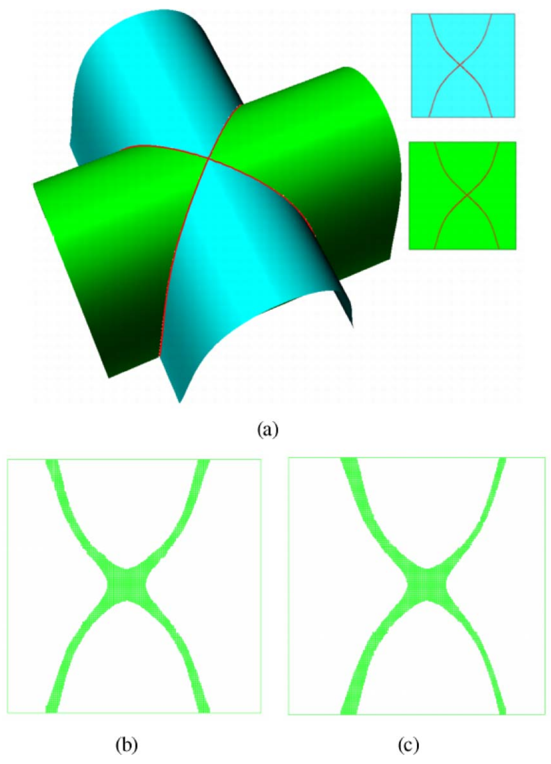


Fig. 7. Two B-spline surfaces intersect with two intersecting curves. (a) Result by our algorithm. (b), (c) Result of the algorithm in [4] in the two parametric domains.

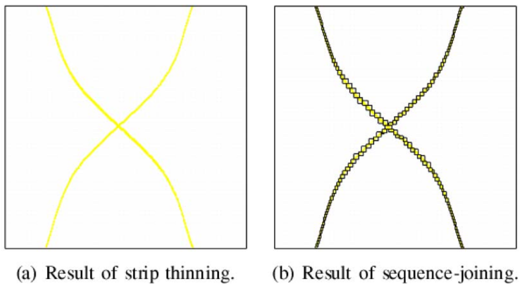

(a) Result of strip thinning.
(b) Result of sequence-joining.

Fig. 8. The strip thinning result (a) and sequence-joining result (b) in generating the two intersection curves which intersect each other.

Moreover, the experimental data for the SISL method [33] are presented in Table 3. The fourth column of Table 3 is the number of intersection curve segments generated by SISL method. The third column is the precision of the intersection curve, which is calculated with (13), after sampling 200 points along each intersection segment. Comparing with our method, the SISL method fails to handle the self-intersection case demonstrated in Fig. 11. As illustrated in Fig. 11b, the SISL method outputs the boundary of the B-spline surface as the self-intersection curve. Moreover, the SISL method can divide an entire intersection curve into subsegments. For instance, in the example of Fig. 7, where two B-spline surfaces intersect with two intersecting curves, the SISL method generates 107 pieces of intersection segments.

In our test, the two B-spline surfaces are both bicubic. The intersection curves are all displayed in red in Figs. 5a, 6a, 7a, 9a, 10a, and 11a, where the larger subfigures are the intersection curves in 3D space, and the two smaller subfigures are the intersection curves on the two parameter domains. Moreover, to clarify the improvement of our algorithm over the one in [4], we illustrate the result of the algorithm in [4] in the two parameter domains in Figs. 5b, 5c, 6b, 6c, 7b, 7c, 9b, 9c, 10b, 10c, and 11b.

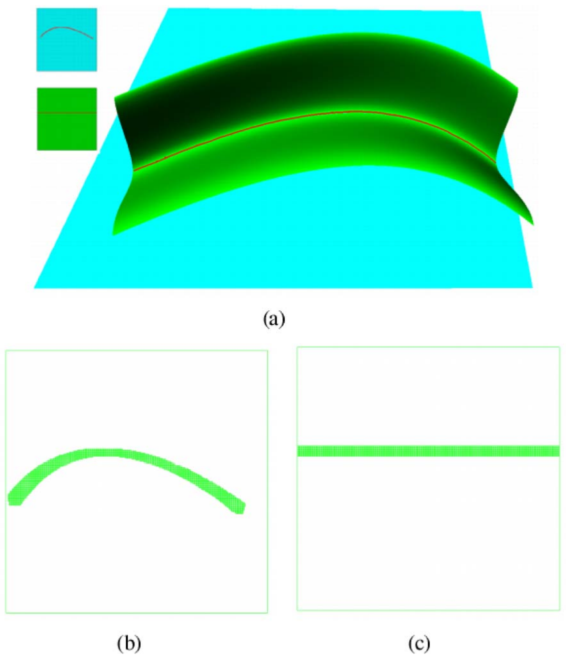


Fig. 9. Tangential intersection. (a) Result by our algorithm. (b), (c) Result by the algorithm in [4] in the two parametric domains.

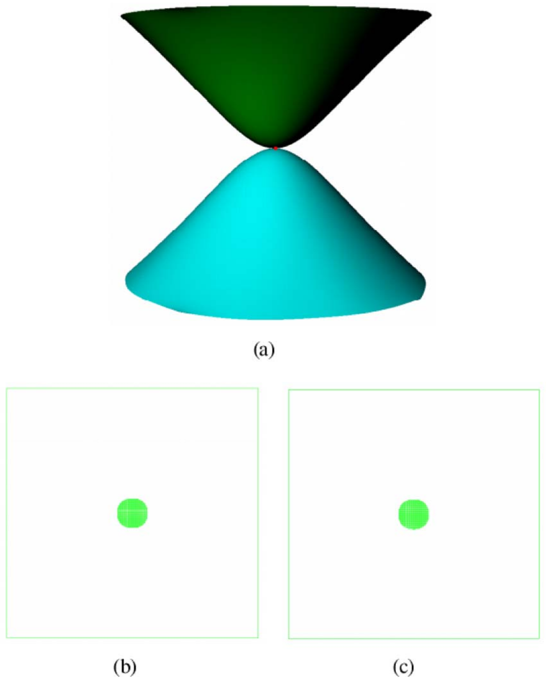


Fig. 10. Two B-spline surfaces intersect at a point. (a) Result by our algorithm. (b), (c) Result by the algorithm in [4] in two parametric domains.

Fig. 5 shows an example of two B-spline surfaces intersecting with one curve. The initial precision of the intersection curve is 8.6×10^{-4}. After only three accuracy improvement iterations, the precision is improved to 3.8×10^{-11}. On the other hand, Fig. 6 illustrates an example of two B-spline surfaces intersecting with two separate curves. While the initial precision of the intersection curve is 5.3×10^{-4}, it is improved to 2.0×10^{-11} after only two accuracy improvement iterations.

Furthermore, in Fig. 7, we illustrate a more complicated example that two B-spline surfaces intersect with two intersecting curves. Fig. 8a shows the result generated by strip thinning algorithm developed in Section 5, which is two intersecting strips. Although the two strips intersect

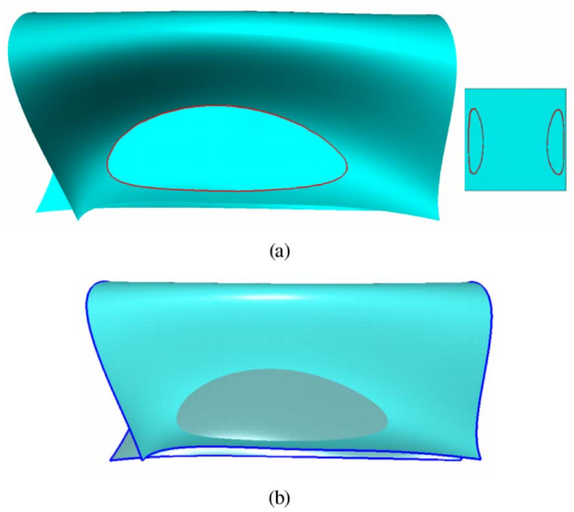


Fig. 11. Self-intersection. (a) Result by our algorithm. (b) The SISL method [33] takes the boundary (in blue) of the surface as the self-intersection.

with each other, the sequence-joining algorithm successfully clusters them as two sequences of rectangles, as illustrated in Fig. 8b. Based on them, two intersecting curves are generated (see Fig. 7). Given high bottom-level resolution $2,048 \times 2,048 \times 2,048 \times 2,048$, the precision of intersection curves reaches 9.3×10^{-6}. It should be pointed out that, while our method generates two pieces of intersection curves, the SISL method [33] produces 107 intersection segments (refer to Table 3). If postprocessing is performed to the SISL result, for example, providing the intersection result to the SISL intersection curve marching algorithm, the segments are expected to be joined to the correct curves.

To illustrate the robustness of the affine arithmetic-based surface intersection algorithm, Fig. 9 presents an example of tangential intersection with high bottom-level resolution, and the precision of the tangential intersection attains 6.1×10^{-6}. Moreover, in Fig. 10, two surfaces are tangent at one point. Our method finds the single intersection point successfully. This example demonstrates the capability of our method in capturing detail intersection.

Additionally, our method can be employed to produce the self-intersection of a surface (see Fig. 11). When computing the self-intersection using our algorithm, we duplicate the given surface and denote them as $\boldsymbol{P}(u,v)$ and $\boldsymbol{Q}(s,t)$, respectively. Moreover, we suppose that the part of the given surface bounded by an AABB and its one-ring adjacent AABBs do not self-intersect. So, we do not perform the overlap test between an AABB and any AABB in its one-ring adjacent AABBs, including itself. Fig. 11 shows an example of self-intersection. Since there is only one surface, we calculate the precision, i.e., relative RMS error (13), for the self-intersection curve with $d_i = d_i(\boldsymbol{P})$. Because the bottom-level resolution is high, the precision of the self-intersection curve meets 5.6×10^{-6}. Note that the SISL method [33] does not have functionality for surface self-intersection, but when the two surfaces intersected are the same the method returns the boundary of the surface to describe the overlap region(see Fig. 11b).

Consequently, according to the comparisons presented above, the practical strategy for developing fast and robust industrial intersection codes would be:

1. Run a preprocess to detect the intersection regions and intersection types, using Sinha's theorem [25], bounding box test, and normal cone test [23], [24], and so on;
2. for those regions with transversal intersections, exploit fast and appropriate methods, such as SISL method [33], and so on;
3. for those regions with singular intersections, for example, tangential intersections and self-intersections, employ the proposed GPU accelerated AA-based method.

Finally, though our method is developed for B-spline surface intersection, it can naturally be extended to deal with NURBS intersection. NURBS is defined on four-dimensional projective space, and its euclidean coordinate is determined by three fractions, i.e., dividing the first three functions by the fourth weight function. Usually, the weights of the NURBS should be greater than zero. Then, the fourth weight function will not contain zero in the domain of the NURBS, and whether the NURBS contains (0,0,0) depends entirely on the numerators of the fractions. Therefore, to extend our method for NURBS intersection, it is just required to replace the right hands of the overlap test equation (12) by the numerators of the differences of two corresponding fractions.

7 Conclusion

In this paper, we presented a GPU-accelerated B-spline surface intersection algorithm based on affine arithmetic. First, we extended the affine arithmetic to calculate the range of the recursively and locally defined B-spline basis functions. Second, the affine arithmetic-based surface intersection algorithm was accelerated by GPU. Finally, we developed efficient methods to thin the strip-shaped intersection regions produced by the affine arithmetic-based intersection algorithm, extracted the intersection points, and improved their accuracy. Due to the acceleration from the GPU, the speed of this algorithm is very fast, achieving real-time response. Moreover, thanks to the reliability of affine arithmetic, the surface intersection algorithm based on affine arithmetic is very robust. Because surface intersection is a fundamental tool in the CAD system, the surface intersection method developed in this paper will have important applications in geometric design.

Acknowledgments

The authors are very grateful to the reviewers for their invaluable advice. This work was supported by the Natural Science Foundation of China (Nos. 61379072, 60970150, and 60933008).

References

[1] C. Hu, T. Maekawa, N. Patrikalakis, and X. Ye, "Robust Interval Algorithm for Surface Intersections," *Computer-Aided Design*, vol. 29, no. 9, pp. 617-627, 1997.

[2] A. Krishnamurthy, R. Khardekar, S. McMains, K. Haller, and G. Elber, "Performing Efficient NURBS Modeling Operations on the GPU," *IEEE Trans. Visualization and Computer Graphics*, vol. 15, no. 4, pp. 530-543, July/Aug. 2009.

[3] M. Gleicher and M. Kass, "An Interval Refinement Technique for Surface Intersection," *Proc. Graphics Interface*, pp. 242-249, 1992.

[4] L. De Figueiredo, "Surface Intersection Using Affine Arithmetic," *Proc. Graphics Interface*, pp. 168-175, 1996.

[5] http://www.nvidia.com/object/cuda_home_new.html, 2013.

[6] R. Moore, *Interval Analysis*. Prentice Hall, 1966.

[7] K. Suffern and E. Fackerell, "Interval Methods in Computer Graphics," *Computers & Graphics*, vol. 15, no. 3, pp. 331-340, 1991.

[8] J. Snyder, "Interval Analysis for Computer Graphics," *ACM SIGGRAPH Computer Graphics*, vol. 26, no. 2, pp. 121-130, 1992.

[9] J. Comba and J. Stol, "Affine Arithmetic and Its Applications to Computer Graphics," *Proc. Brazilian Symp. Computer Graphics and Image Processing (SIBGRAPI '93)*, pp. 9-18, 1990.

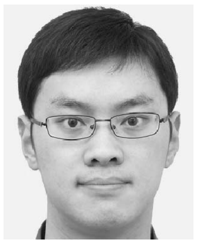
[10] H. Shou, R. Martin, I. Voiculescu, A. Bowyer, and G. Wang, "Affine Arithmetic in Matrix Form for Polynomial Evaluation and Algebraic Curve Drawing," *Progress in Natural Science*, vol. 12, no. 1, pp. 77-81, 2002.

[11] H. Shou, H. Lin, R. Martin, and G. Wang, "Modified Affine Arithmetic is More Accurate than Centered Interval Arithmetic or Affine Arithmetic," *Proc. IMA Int'l Conf. Math. of Surfaces*, pp. 355-365, 2003.

[12] H. Shou, H. Lin, R. Martin, and G. Wang, "Modified Affine Arithmetic in Tensor Form for Trivariate Polynomial Evaluation and Algebraic Surface Plotting," *J. Computational and Applied Math.*, vol. 195, nos. 1/2, pp. 155-171, 2006.

[13] T. Sederberg and R. Farouki, "Approximation by Interval Bézier Curves," *IEEE Computer Graphics and Applications,* vol. 12, no. 5, pp. 87-95, Sept. 1992.
[14] C. Hu, T. Maekawa, E. Sherbrooke, and N. Patrikalakis, "Robust Interval Algorithm for Curve Intersections," *Computer-Aided Design,* vol. 28, nos. 6/7, pp. 495-506, 1996.
[15] H. Mukundan, K. Ko, T. Maekawa, T. Sakkalis, and N. Patrikalakis, "Tracing Surface Intersections with Validated ODE System Solver," *Proc. Ninth ACM Symp. Solid Modeling and Applications,* pp. 249-254, 2004.
[16] N. Patrikalakis, T. Maekawa, K. Ko, and H. Mukundan, "Surface to Surface Intersection," *Proc. Int'l CAD Conf. and Exhibition,* vol. 4, 2004.
[17] N. Aziz, R. Bata, and S. Bhat, "Surfaces: Bezier Surface/Surface Intersection," *IEEE Computer Graphics and Applications,* vol. 10, no. 1, pp. 50-58, Jan. 1990.
[18] D. Manocha and J. Canny, "A New Approach for Surface Intersection," *Int'l J. Computational Geometry and Applications,* vol. 1, no. 4, pp. 491-516, 1991.
[19] S. Krishnan and D. Manocha, "An Efficient Surface Intersection Algorithm Based on Lower-Dimensional Formulation," *ACM Trans. Graphics,* vol. 16, no. 1, pp. 74-106, 1997.
[20] D. Manocha and S. Krishnan, "Algebraic Pruning: A Fast Technique for Curve and Surface Intersection," *Computer Aided Geometric Design,* vol. 14, no. 9, pp. 823-845, 1997.
[21] G. Renner and V. Weiss, "Exact and Approximate Computation of B-Spline Curves on Surfaces," *Computer-Aided Design,* vol. 36, no. 4, pp. 351-362, 2004.
[22] D. Lasser, "Intersection of Parametric Surfaces in the Bernstein-Bézier Representation," *Computer-Aided Design,* vol. 18, no. 4, pp. 186-192, 1986.
[23] S. Briseid, T. Dokken, T. Hagen, and J. Nygaard, "Spline Surface Intersections Optimized for GPUs," *Proc. Sixth Int'l Conf. Computational Science (ICCS '06),* pp. 204-211, 2006.
[24] S. Briseid, T. Dokken, and T.R. Hagen, "Heterogeneous Spline Surface Intersections," *Proc. 26th Spring Conf. Computer Graphics (SCCG '10),* pp. 141-148, http://doi.acm.org/10.1145/1925059.1925085, 2010.
[25] P. Sinha, E. Klassen, and K. Wang, "Exploiting Topological and Geometric Properties for Selective Subdivision," *Proc. First Ann. Symp. Computational Geometry,* pp. 39-45, 1985.
[26] D. Alcantara, A. Sharf, F. Abbasinejad, S. Sengupta, M. Mitzenmacher, J. Owens, and N. Amenta, "Real-Time Parallel Hashing on the GPU," *ACM Trans. Graphics,* vol. 28, no. 5, article 154, 2009.
[27] J. Hoschek and D. Lasser, *Fundamentals of Computer-Aided Geometric Design.* AK Peters, 1993.
[28] T. Dokken, "Aspects of Intersection Algorithms and Approximation," Doctor thesis, Univ. of Oslo, 1997.
[29] L. De Figueiredo and J. Stolfi, "Affine Arithmetic: Concepts and Applications," *Numerical Algorithms,* vol. 37, no. 1, pp. 147-158, 2004.
[30] R. Martin, H. Shou, I. Voiculescu, A. Bowyer, and G. Wang, "Comparison of Interval Methods for Plotting Algebraic Curves," *Computer Aided Geometric Design,* vol. 19, pp. 553-587, 2002.
[31] N. Patrikalakis and T. Maekawa, *Shape Interrogation for Computer Aided Design and Manufacturing.* Springer-Verlag, 2002.
[32] H. Lin, W. Chen, and G. Wang, "Curve Reconstruction Based on an Interval B-Spline Curve," *The Visual Computer,* vol. 21, no. 6, pp. 418-427, 2005.
[33] http://www.sintef.no/sisl, 2013.

Hongwei Lin received the BSc degree from the Department of Applied Mathematics at Zhejiang University, China, in 1996, and the PhD degree from Department of Mathematics at Zhejiang University in 2004. He worked as a communication engineer from 1996 to 1999. He is an associate professor in the Department of Mathematics, State Key Laboratory of CAD&CG, Zhejiang University. His current research interests include geometric design, computer graphics, and computer vision. He is a member of the IEEE.



Yang Qin received the bachelor's degree from Wuhan University in 2009, and the master's degree from the State Key Laboratory of CAD&CG, Zhejiang University, in 2012. He is currently working in Microsoft China. His research interests include geometric design and parallel computing.

Hongwei Liao received the bachelor's degree from Nanchang University in 2010, and the master's degree from the State key Laboratory of CAD&CG, Zhejiang University, China, in 2013. He is currently working in Software Development Department of China Construction Bank. His research interests include mesh generation and high-performance computing.

Yunyang Xiong received the bachelor's degree from Suzhou University in 2011. He is currently a graduate student in the State Key Laboratory of CAD&CG, Zhejiang University, China. His research interests include geometric design, computer graphics, and geometric iterative methods.