Computer-Aided Design 58 (2015) 132–140

Contents lists available at ScienceDirect

Computer-Aided Design

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

Efficient direct rendering of deforming surfaces via shared subdivision trees☆

Fuchang Liu [a,c,*], Tobias Martin [b], Sai-Kit Yeung [c], Markus Gross [b]

[a] Hangzhou Normal University, Hangzhou, China
[b] ETH Zurich, Switzerland
[c] Singapore University of Technology and Design, Singapore

HIGHLIGHTS

- We present Shared Subdivision Trees (SST) to rasterize implicit surfaces on GPUs.
- We address the problem of efficiently rendering implicit surfaces which undergo a nonlinear deformation throughout the rendering process.
- We map Shared Subdivision Trees well to parallel computing platforms such as CUDA.

ARTICLE INFO

Keywords:
Isosurface visualization
GPU rendering
Computational geometry and object modeling

ABSTRACT

In this paper, we present a subdivision-based approach to rasterize implicit surfaces embedded in volumetric Bézier patches undergoing a nonlinear deformation. Subdividing a given patch into simpler patches to perform the surface rasterization task is numerically robust, and allows guaranteeing visual accuracy even in the presence of geometric degeneracies. However, due to its memory requirements and slow convergence rates, subdivision is challenging to be used in an interactive environment. Unlike previous methods employing subdivision, our approach is based on the idea where for a given patch only one subdivision tree is maintained and shared among pixels. Furthermore, as the geometry of the object changes from frame to frame, a flexible data structure is proposed to manage the geometrically varying Bézier patches. The resulting algorithm is general and maps well to parallel computing platforms such as CUDA. We demonstrate on a variety of representative graphics and visualization examples that our GPU scheme scales well and achieves up to real-time performance on consumer-level graphics cards by guaranteeing visual accuracy.

1. Introduction

Deforming *B*-spline volumes with embedded scalar fields frequently occur in a variety of computer graphics and engineering applications. For instance, in free-form deformation [1] an implicit surface of a scalar field is deformed by deforming the geometry of its associated *B*-spline bounding volume. *B*-spline volumes are also a fundamental primitive in isogeometric analysis [2] where they are used to represent the geometry of a physical object. Physical analysis is applied directly to the *B*-spline volume representation, where the analysis result is represented as an associated attribute. Depending on the simulation scenario, the geometry of the representation may undergo shape changes. For instance, an elastic body deforms when external forces are applied, where stress is an attribute of the deforming object.

In this paper, we address the problem of efficiently rendering implicit surfaces which undergo a nonlinear deformation throughout the rendering process. The deformation is performed on a volumetric representation, which can be converted into a set of Bézier volumes. While the topology of the deforming surface may remain the same throughout the animation, its scale may change nonuniformly from frame to frame. Extraction- and sampling-based methods are not only challenged by the changing surface properties and the dynamic volumetric deformations, but also by the reconstruction of all the features present in the implicit surface (for instance, see thin features in Fig. 1).

☆ This paper has been recommended for acceptance by Dr. Vadim Shapiro.

* Correspondence to: Digital Media and HCI Research Center, Hangzhou Institute of Service Engineering, Hangzhou Normal University, Yuhang Qu Haishu Road 58, Hangzhou 311121, China.

E-mail addresses: liufububai@gmail.com, liufu@ewha.ac.kr (F. Liu), martint@inf.ethz.ch (T. Martin), saikit@sutd.edu.sg (S.-K. Yeung), grossm@inf.ethz.ch (M. Gross).

http://dx.doi.org/10.1016/j.cad.2014.08.005
0010-4485/

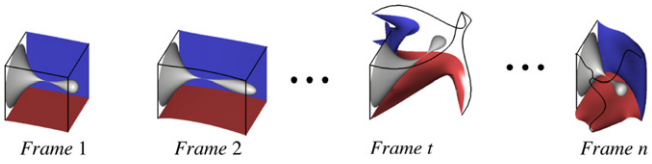

Fig. 1. Deformation sequence of an isosurface of a tri-quintic algebraic function. Methods to extract the implicit surface, such as Marching Cubes, are challenged because of the nonlinear distortions and thin surface features.

Given this scenario, subdividing the Bézier patches into simpler patches is key. Traditionally, a subdivision-based approach builds a subdivision tree for each pixel, where sub-patches in the tree are kept and only subdivided further if they potentially contain a piece of the surface overlapping with the pixel. In the limit, the leafs of the subdivision tree constitute to that piece of the surface, passing through the pixel. However, subdivision is computationally expensive and only converges linearly to a solution. Therefore, instead of subdividing the patch to pixel size, it is only subdivided until all intersections can be determined using the Newton–Raphson method. Then, the local subdivision tree is discarded. In order to reduce the size of local sub-division trees, a pre-subdivision stage [3] is employed: before rendering takes place, the patch is first subdivided into a set of simpler patches. The main drawback with this strategy is that a hierarchical data structure has to be maintained, and has to be rebuilt whenever the geometry changes. This poses additional challenges to map such a scheme efficiently to GPU. Furthermore, the hierarchy is view independent, i.e., it may consist of too many (or too few) levels, and also patches which are occluded from the current view.

Main contribution: we present a novel concept for GPU, called *Shared Subdivision Trees* (SST), to rasterize implicit surfaces represented by multiple Bézier patches undergoing a nonlinear deformation during rendering. Conceptually, as illustrated in Fig. 2, for a given patch, a single subdivision tree is maintained which is shared among pixels. A patch in the subdivision tree is only subdivided further if requested by a screen pixel, which eliminates the redundant subdivision work. The proposed method can be seen as moving the pre-subdivision stage into the rendering stage, where the subdivision tree of a given patch is built by exploiting the high parallelism of current GPUs. The visibility problem is solved by a conventional sweep and prune method which allows to handle datasets as they occur in practice. We demonstrate on a variety of representative examples that our scheme is computationally efficient and yields interactive and for some examples even real-time frame rates. In addition, we verify that our scheme scales well with respect to memory requirement and rendering speed.

The outline of this paper is as follows. After discussing the related work in Section 2, the mathematical framework used for this work is introduced in Section 3. The proposed algorithm is described in Section 4 and its implementation is discussed in Section 5. Then, three applications and associated studies are presented in Section 6.1. Finally, we evaluate the efficiency of our proposed method in Section 6.2, and conclude the paper in Section 7.

2. Previous work

Direct rendering on uniform grids: there is a vast body of work to directly render implicit surfaces of volumetric scalar fields. A variety of highly efficient methods exist when the scalar field in world space can be described by a trivariate or piecewise trivariate polynomial. Methods in this category date back to the work by Rockwood [4] which computes univariate contours from a Bézier volume. Roots of these contours correspond to points on the isosurface. A related approach presented in [5] converts the algebraic function along the ray into Bernstein form to efficiently and robustly determine all intersections between the ray and the implicit surface. Knoll et al. [6] present an approach to render implicit surfaces of algebraic functions using interval arithmetic achieving real-time frame rates. Approaches falling into the same category, but which are based on sampled volume data are [7–9]. More recently, Liu et al. [10] present an isosurface rasterization approach exploiting cache coherency to further speed up rendering.

Due to the polynomial nature of the scalar field, the scalar field along a viewing ray can be represented in closed form. This property results in a lower memory footprint making it easier to solve the problem on the GPU. However, in our scenario, the volume embedding the scalar field undergoes a nonlinear deformation (Fig. 1 and for more examples Figs. 7 and 8). In this case, the scalar field consists of a highly nonlinear term which makes it impossible to express the scalar function along the ray analytically. Because of that, it is unclear how to extend the methods above to also work efficiently in this scenario. In this paper, we present an efficient rasterization method which robustly renders isosurfaces embedded in deformed objects.

Extraction-based methods: among the first methods to render scalar data embedded in deformed volumes is [11]. The method is based on an isosurface sampling approach similar to [12]: points are iteratively projected onto the isosurface and the surface is rendered using a point-based rendering system such as [13]. High visual accuracy can be achieved following this strategy. However, determining a point sampling which guarantees visual accuracy is difficult. These methods generally tend to oversample the implicit surface in order to reconstruct thin or smaller features as the one shown in Fig. 2. Extraction based methods face similar problems. Such an approach first extracts the implicit surface using a method such as Marching Cubes [14], or Marching Tetrahedra [15]. Then, the extracted triangle mesh approximating the smooth implicit surface is rendered. While extraction can be executed very efficiently, the smooth representation first has to be discretized into a linear format. This requires the sampling of the volumetric patch. Efficiently generating a sampling such that the extracted triangle mesh is accurate up to image resolution is an open problem. Note that, all these challenges are amplified when the implicit surface undergoes a nonlinear deformation every frame.

Ray-sampling-based methods: given a ray passing through a pixel and a deformed volumetric patch with embedded scalar field, an intersection of the ray with the implicit surface is computed in two steps: (1) determine the entry and exit point of the ray into the patch; and (2) perform root finding on these bounds to identify where the ray intersects the implicit surface. Since the scalar function cannot be written in closed form, as discussed above, the latter step requires sampling of the scalar field along the ray, where for each sample, the inverse function has to be evaluated using a numerical method. For instance, [16,17] adaptively sample this function to compute a polynomial interpolant based on a Legendre basis which can be arbitrarily close to the solution. Similarly, [18] present a GPU ray-caster using a frequency based adaptive sampling approach to account for high variations along the ray. To achieve interactive frame rates, the method stores the volumetric patches in a grid.

Ray-sampling methods, such as the ones discussed above, assume that the mapping between the reference element to the deformed patch is bijective. However, this is often not the case in practice. For instance, in physically based animation [19], patches undergoing a nonlinear deformation may self-intersect or even invert. This results in zero Jacobians, where at these locations the mapping is not bijective and therefore, a numerical method such as Newton–Raphson to compute the inverse cannot be used. This type of data presents severe stability and convergence issues for the rendering approaches mentioned above. Furthermore, boundaries

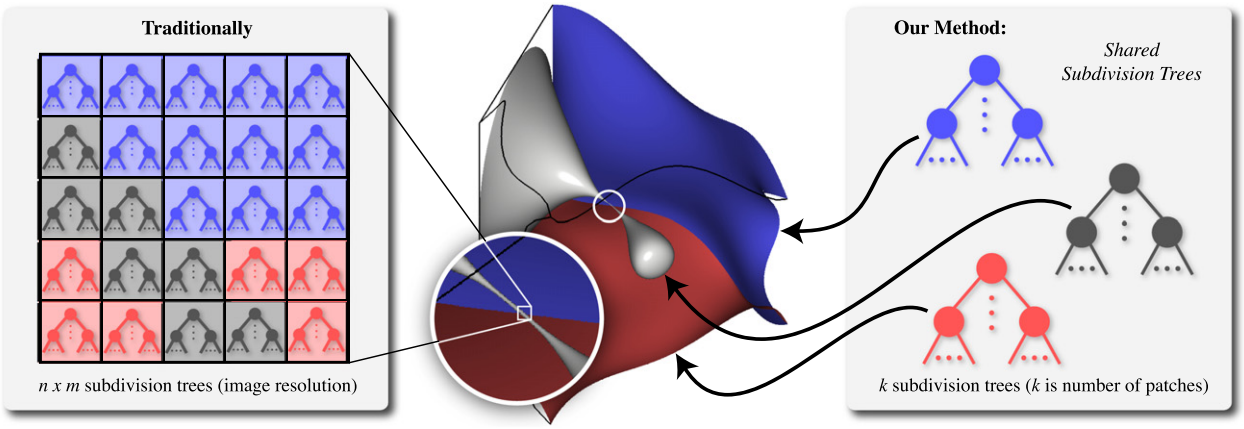

Fig. 2. Subdivision and numerical root-finding allow to robustly render implicit surfaces without missing features. Traditionally, the problem has been approached in parallel on the pixel level. Our method efficiently maps to the GPU by sharing a single subdivision tree for a given patch among multiple pixels.

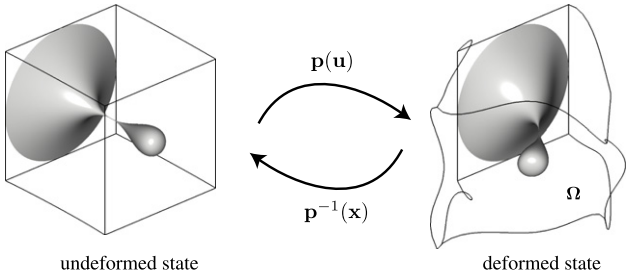

Fig. 3. Nonlinear deformation of a tri-quintic Bézier volume. The map also deforms an isosurface of an embedded scalar field. The inverse $\mathbf{p}^{-1}(\mathbf{x})$ is highly nonlinear.

of volumetric patches are often degenerate, e.g., elements along a cylindrical axis, which complicates the computation of entry and exit points. Our proposed method robustly handles these types of scenarios.

The introduction of geometric constraint solvers in the pioneering work [20] is based on subdivision and numerical root-finding, and thus guarantee to find all values satisfying a system of nonlinear geometric constraint equations. The method proposed in [21] builds up on this framework: for each pixel, a system of three nonlinear equations is solved to determine intersections with the implicit surface. The method results in a tremendous memory overhead, since it constructs a subdivision tree for each pixel independently. Because of this property, it is difficult to map this scheme to the GPU. In this paper, we reformulate the problem in such a way, that the intersection problem can be efficiently solved on a GPU. At the same time, it retains the guarantees of a geometric constraint solver. The key of the proposed method is to share both, the subdivision work and the resulting patches across viewing pixels to robustly compute the implicit surface intersections.

In the next section, we introduce the mathematical framework on which we base our method. It also includes a discussion of classical ray/isosurface intersection given this type of data as a rationale to propose Shared Subdivision Trees and its implementation to fully exploit the parallelism of consumer level GPUs in Sections 4 and 5.

3. Mathematical background

The input to our method is a volumetric representation with embedded scalar field undergoing a nonlinear deformation in discrete time steps, where we assume that the representation can be converted into a set of Bézier volumes. In this work each frame is treated independently, i.e., calculations of a previous time step are not used in the current time step. In the following paragraphs we solely focus on rendering implicit surfaces embedded in the Bézier volume of a single time step. Fig. 3 illustrates the formalities made in the following two paragraphs where for illustrative purposes the input consists of a single rational Bézier volume of degree (l, m, n), formulated as

$$\mathbf{V}(\mathbf{u}) = \frac{\sum_{i=0}^{l}\sum_{j=0}^{m}\sum_{k=0}^{n} w_{ijk}\,\mathbf{c}_{ijk}\,\theta_i^l(u)\,\theta_j^m(v)\,\theta_k^n(w)}{\sum_{i=0}^{l}\sum_{j=0}^{m}\sum_{k=0}^{n} w_{ijk}\,\theta_i^l(u)\,\theta_j^m(v)\,\theta_k^n(w)}. \tag{1}$$

Here, the parameter $\mathbf{u} = \{u, v, w\}$ lives in a cubic parameter domain, where $\theta_i^n(u)$ is the ith Bernstein polynomial [22] of degree n. $\mathbf{c}_{ijk}$ define control points of a $(l+1) \times (m+1) \times (n+1)$ control grid, where, $\mathbf{c}_{ijk} = \{x_{ijk}, y_{ijk}, z_{ijk}, a_{ijk}\}$. The first three components of $\mathbf{c}_{ijk}$ represent world space positions, and the fourth component is a scalar attribute. Thus, $\mathbf{V}(\mathbf{u})$ can be separated into two mappings, i.e., $\mathbf{V}(\mathbf{u}) = \{\mathbf{p}(\mathbf{u}), \alpha(\mathbf{u})\}$, where $\mathbf{p}(\mathbf{u})$ deforms the unit cube into $\Omega \in \mathbb{R}^3$, with $\alpha(\mathbf{u})$ as its associated scalar volume (see Fig. 3).

The inverse $\mathbf{p}^{-1}(\mathbf{x})$ maps a point $\mathbf{x} \in \Omega$ back to the unit cube. Given a user-specified isovalue $\hat{a}$, an implicit surface residing within the bounds of Ω is defined as the set $\mathcal{S} = \{\mathbf{x} \mid \alpha(\mathbf{p}^{-1}(\mathbf{x})) = \hat{a}, \mathbf{x} \in \Omega\}$. Due to the rational nature of $\mathbf{p}(\mathbf{u})$, its inverse $\mathbf{p}^{-1}(\mathbf{x})$ cannot be expressed in closed form. Hence, the inverse of a given point $\mathbf{x} \in \Omega$ can at most be numerically approximated by a value $\mathbf{u}^*$ such that $|\mathbf{p}^{-1}(\mathbf{x}) - \mathbf{u}^*| < \epsilon$, where ϵ is sufficiently small. Newton's method is generally used to improve the accuracy of $\mathbf{u}^*$. However, if a local self-intersection of the Bézier volume is crossed during a Newton iteration, the method does not succeed.

3.1. Classical ray/isosurface intersection

Ray/isosurface algorithms generally give answer to the question, at which point a ray in world space intersects $\mathcal{S}$. Such an intersection point belongs to $\mathcal{S}$ if it satisfies three constraint equations, where the first equation is $\alpha(\mathbf{u}) - \hat{a} = 0$. Furthermore, the point also has to sit along the ray: by following the development of [23], the ray can be represented as the intersection of two orthogonal planes, $\langle \mathbf{x}, \mathbf{n}_i \rangle + d_i = 0$, for $i = 0, 1$, where $\mathbf{n}_i$ is the normal of plane i. The patch $\mathbf{p}(\mathbf{u})$ is substituted into these two equations, yielding the second and the third constraint equations, $\langle \mathbf{p}(\mathbf{u}), \mathbf{n}_i \rangle + d_i = 0$, for $i = 0, 1$.

Given these three equations, a new Bézier volume, $\mathbf{q}(\mathbf{u})$, with control points $\{\langle \mathbf{x}_{ijk}, \mathbf{n}_0 \rangle + d_0, \langle \mathbf{x}_{ijk}, \mathbf{n}_1 \rangle + d_1, a_{ijk} - \hat{a}\}$, where $\mathbf{x}_{ijk} = \{x_{ijk}, y_{ijk}, z_{ijk}\}$ with associated weights w_{ijk} can be constructed. The point where the ray intersects the isosurface can now be

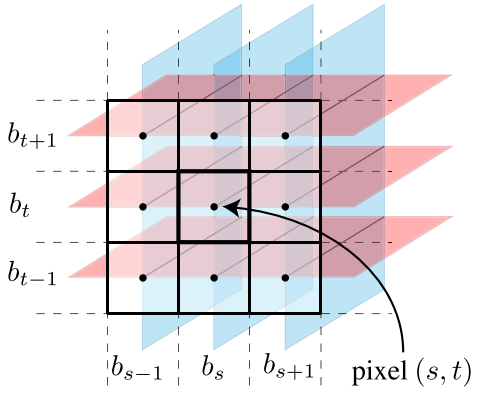

Fig. 4. 3×3 pixel neighborhood on the near plane. Planes (red and blue) whose intersections coincide with pixel centers are defined as geometric constraints. (For interpretation of the references to colour in this figure legend, the reader is referred to the web version of this article.)

determined by solving $\mathbf{q}(\mathbf{u}) = \{0, 0, 0\}$ for $\mathbf{u}$. This can be robustly achieved by splitting the Bézier volume $\mathbf{q}(\mathbf{u})$ into smaller subpatches via Bézier subdivision [24] by only keeping those subpatches which contain the origin. Martin et al. [21] combine subdivision and numerical root finding to determine all solutions along the ray following this strategy.

This scheme is general: given *any* ray in world space, the intersection problem can be solved robustly. This generality is especially useful for secondary rays in ray tracing solutions (e.g., [25]). However, it comes at the price that for each ray an independent problem has to be solved, i.e., construct an individual subdivision tree for each ray in order to faithfully determine all the intersection points. The pioneering work [26], and the famous Bézier Clipping method [27] follow a very similar strategy to directly render parametric surfaces. While local memory might be sufficient to render parametric surfaces, mapping such an approach to the fragment level of a GPU to render isosurfaces of deformed volume data is generally not feasible, because local subdivision trees quickly exceed the local memory size.

The following section arrives at a solution which rasterizes the isosurface as seen from the eye, where only a single subdivision tree for a given Bézier volume is maintained and shared among the pixels. This results in a scheme which requires significantly less memory and thus allowing for the efficient implementation and execution on a GPU.

4. Shared subdivision trees for isosurface rasterization

Given the mathematical framework introduced above, in this section we propose our isosurface rasterization algorithm to accurately render the isosurface $\mathcal{S}$ of a given time step from the current camera position. While our method will be applied to objects which are represented as a union of Bézier volumes, for simplicity of discussion, we focus here on a single Bézier volume $\mathbf{p}(\mathbf{u})$. For that we assume that its world space coefficients have been transformed by the camera and projection matrix into perspective space. Hence, the isosurface $\mathcal{S}$ lies within the transformed camera frustum, a cube of size $[-1, 1]^3$. The computation discussed in this section is conducted on its front side, corresponding to the camera's near plane onto which we impose a $n \times n$ image.

By referring to Fig. 4, given pixel (s, t), the goal is to determine all points which lie on the isosurface $\mathcal{S}$ and within pixel (s, t). The first constraint is described with the inequality $|\alpha(\mathbf{u}) - \hat{a}| \leq \epsilon$, where in our implementation $\epsilon = 10^{-5}$. The requirement that a point has to lie within the pixel can be described with two constraint inequalities: analogously to the strategy in Section 3.1, given half the pixel width p_w, the two scalars, $b_s = s/n + p_w$ and $b_t = t/n + p_w$, are used to define two perpendicular planes, $x - b_s = 0$ and $y - b_t = 0$, coinciding with the center of the pixel. Note that the first plane passes through all the pixel centers of the image column s, and the second plane passes through all pixel centers of row t. Substituting patch $\mathbf{p}(\mathbf{u}) = \{x(\mathbf{u}), y(\mathbf{u}), z(\mathbf{u})\}$ into the two plane equations yields the second and third constraint inequalities, $|x(\mathbf{u}) - b_s| \leq p_w$ and $|y(\mathbf{u}) - b_t| \leq p_w$, respectively.

Given these three constraint inequalities, a new Bézier volume with control points $\{x_{ijk} - b_s, y_{ijk} - b_t, a_{ijk} - \hat{a}\}$ with corresponding weights w_{ijk} can be constructed. This Bézier volume is equivalent to $\mathbf{s}(\mathbf{u}) - \mathbf{o}_{st}$, where $\mathbf{s}(\mathbf{u})$ is a Bézier volume with control points $\{x_{ijk}, y_{ijk}, a_{ijk}\}$ with the same weights as $\mathbf{p}(\mathbf{u})$, and $\mathbf{o}_{st} = \{b_s, b_t, \hat{a}\}$. All parameters $\mathbf{u}^*$ satisfying these constraints can now be determined by solving $|\mathbf{s}(\mathbf{u}) - \mathbf{o}_{st}| \leq \{p_w, p_w, \epsilon\}$ for $\mathbf{u}$, i.e., $\mathbf{p}(\mathbf{u}^*)$ lies within the pixel and is sufficiently close to the isosurface. Note that the z component of $\mathbf{p}(\mathbf{u}^*)$ is the pixel depth. The offset $\mathbf{o}_{st}$ can be seen as translating the Bézier volume $\mathbf{s}(\mathbf{u})$ into the local coordinate system of pixel (s, t), where the pixel center serves as origin. In this formulation, the patch $\mathbf{s}(\mathbf{u})$ is independent of the pixel offset, i.e., the patch $\mathbf{s}(\mathbf{u})$ can be used by other pixels to solve its individual intersection problems.

This suggests an algorithm, called the *Shared Subdivision Tree* (SST) algorithm, which is based on building only one subdivision tree for a given patch $\mathbf{s}(\mathbf{u})$. Initially a subdivision tree consists of the input patch only. This patch is subdivided if required by a pixel. However, since the tree is independent of a given pixel, subdivided patches can be used to determine the root of other pixels by applying the respective offset as discussed above. Semantically, we refer to this behavior as *sharing* a subdivision tree among pixels, because the relevant patches are shared among various pixels and subdivision work is not performed for each pixel independently. Therefore, this algorithm can be seen as solving the intersection problem globally, which is in contrast to the traditional ray/isosurface intersection discussed in Section 3.1, where the intersection problem is solved independently for each pixel (s, t). In the following we describe the SST algorithm in more detail.

4.1. SST algorithm

The inputs to our algorithm are the screen pixels (s, t), and a list $\mathcal{L}$ consisting of Bézier volumes, constructed as discussed in the previous section. $\mathcal{L}_i$ corresponds to the ith patch in $\mathcal{L}$. The algorithm outputs a solution array $\mathcal{R}$ of parameter values for each pixel. Here, $\mathcal{R}_{st}$ contains the solutions for pixel (s, t). Given that, the following three stages are executed until all intersections have been determined.

(1) Overlap Stage. Each pixel (s, t) is tested whether it overlaps with bounding boxes of the patches in $\mathcal{L}$. If there is an overlap between bounding box of patch $\mathcal{L}_i$ and pixel (s, t), the isosurface may overlap with the pixel as well. Therefore, the tuple (s, t, i), where i refers to patch $\mathcal{L}_i$, is added to pixel index list $\mathcal{I}$.

(2) Solution Stage. Newton's method is used on each tuple in $\mathcal{I}$ to solve equation $|\mathcal{L}_i(\mathbf{u}) - \mathbf{o}_{st}| \leq \{p_w, p_w, \epsilon\}$, for parameter $\mathbf{u}$, as discussed in the previous section. If Newton succeeds, the tuple is removed from $\mathcal{I}$ and the solution $\mathbf{u}$ is added to $\mathcal{R}_{st}$. If all tuples have been removed from $\mathcal{I}$, the algorithm terminates. Otherwise it proceeds to Stage 3.

(3) Subdivision Stage. For all the remaining tuples in $\mathcal{I}$, patches are subdivided and $\mathcal{L}$ is overwritten with the new patches, where patch indices in the tuples are adjusted accordingly. Subpatches which do not have a sign change in their scalar attribute are discarded, as this indicates that they cannot contain a piece of the implicit surface. Go back to Stage 1.

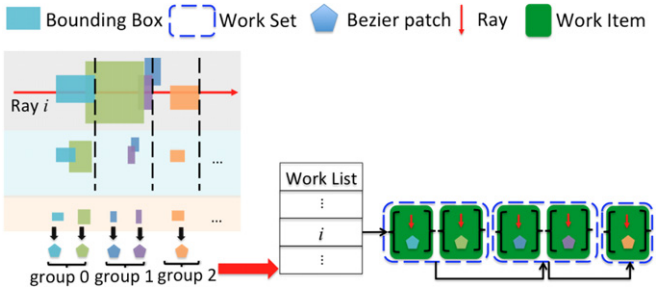

Fig. 5. Illustration of the SST initialization part.

After the termination of this loop, in a final step, each $\mathcal{R}_{st}$ is sorted according to the respective pixel depth. For more details, pseudocode of this algorithm is provided in Appendix A. Since all roots for a given pixel are computed, the method can be used to realize independent transparency and to render CSG objects as discussed in Section 6.1. Note that, each stage of this algorithm can be executed in parallel.

5. Implementation

In the following, we give a brief description on how to implement the above algorithm on existing computing platforms such as CUDA [28]. The initialization part and core of our rendering algorithm are illustrated in Figs. 5 and 6. Here we assume the more general case where multiple input patches are given. All steps are executed through GPU primitives such as sort, compact, and scan, where we use the implementation provided in the Thrust [29] library.

In the initialization part of our implemented rendering framework, we first determine the patches whose bounding boxes are intersected by a given pixel. These patches are sorted along the ray passing through the pixel's center, because as with most existing methods, once the closest intersection to the near plane has been found, the pixel can be removed from the list. We call this grouping process *sweep and group* (SAG). To sort patches along a ray, a data structure such as a Kd-tree [30] could be employed as well. However, those data structures are generally challenged when dealing with dynamic geometry. SAG is a reasonable compromise given the dynamic environment, where the shape of patches changes from frame to frame. As illustrated in Fig. 5, we spawn a thread for each pixel, sweeping along the ray passing through the pixel's center in parallel. Patches are grouped according to the distance from the eye to its corresponding bounding boxes. A patch and its corresponding ray are stored in one work item. Work items corresponding to the same ray are packed into one work set. After performing SAG, work sets are compacted and stored in global GPU memory as a linear list.

The core of our implemented rendering framework corresponds to Section 4.1. As illustrated in Fig. 6, we fetch the work sets and test the overlap for each work item in the *overlap kernel*. Patches which overlap with the pixel are sent to the *solution kernel*. This kernel executes Newton's method to determine roots. If a root for a given pixel has been found, the pixel is flagged in a stencil buffer. If no root has been found, the patch is sent to the *subdivision kernel* to continue subdividing the respective patch. The *compaction kernel* removes those work items which do not pass the *compute overlap kernel* or the *solution kernel*. The detailed GPUs implementation of the three kernels is described in the following:

Overlap kernel. Update the indices in $\mathcal{I}$ to reflect the new indices of the subdivided volumes, and remove those tuples (s, t, i), where the pixel (s, t) does not overlap with the bounding box of the subdivided patch $\mathcal{L}_i$.

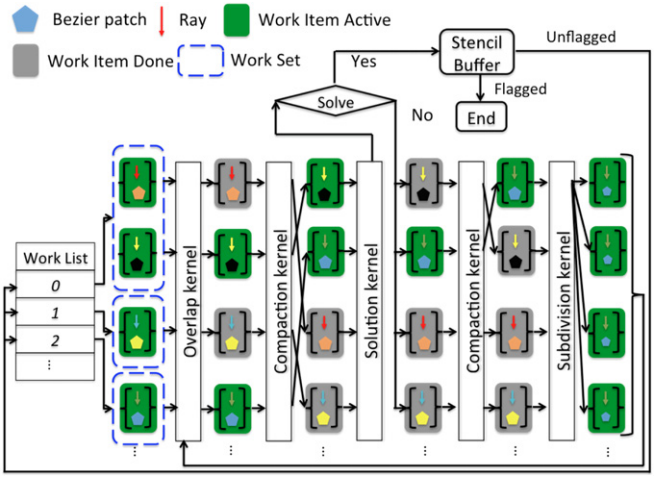

Fig. 6. Illustration of the SST core execution kernels.

Solution kernel. Initialize a *pixel flag list* of the same size as the pixel index list $\mathcal{I}$ with *false*, and a *patch score list* of the same size as $\mathcal{L}$ with 0. Then, for each tuple $(s, t, i) \in \mathcal{I}$, if $\mathcal{L}_i$ contains at most one solution, construct the offset $\mathbf{o}_{st} = \{b_s, b_t, \hat{a}\}$ and execute Newton's method to solve $\mathcal{L}_i(\mathbf{u}) = \mathbf{o}_{st}$. If it succeeds and the computed solution is within the extents of pixel (s, t) and within ϵ distance to the isosurface, add it to the solution vector $\mathcal{R}_{st}$. If it fails, set the flag for pixel (s, t) to *true* and increment the score for patch $\mathcal{L}_i$ by 1.

Subdivision kernel. This kernel consists of 4 steps: (1) iterate through all pixel flags, and remove the tuple from $\mathcal{I}$, if the corresponding flag is *false*. (A tuple (s, t, i) is only removed if a solution in the *solution stage* has been found;) (2) initialize a *patch flag list* of the same size as the patch list $\mathcal{L}$ with *false*. Set those flags to *true* whose corresponding score in the score list is >0; (3) iterate through all patch flags and remove those patches from $\mathcal{L}$ whose corresponding flags are false. (A patch $\mathcal{L}_i$ is only removed if no pixel fails in finding a solution in step (3);) (4) iterate through $\mathcal{L}$, and split each $\mathcal{L}$ in the middle into 8 sub-patches using Bézier subdivision and add them to a new list $\bar{\mathcal{L}}$. Once subdivision has been performed, overwrite $\mathcal{L}$ with $\bar{\mathcal{L}}$.

Newton's method fails when the Jacobian of the mapping is ill-conditioned, or when the solution falls outside the parameter domain of the patch. In this case, the patch requires further subdivision. A normal cone test to find out whether at most one root is within a given patch is applied as in [21]. Bounding boxes are oriented such that they tightly fit the patch as it is done in the same work. An important aspect of our algorithm is that it maintains global arrays for patches and pixels, where all available threads assist in the computation. Related rendering methods implemented on the local memory where each pixel is treated independently in parallel, might have more efficient memory access than a method which is based on global memory. However, the high usage of local memory may add a bottleneck to GPU kernels resulting in decreased GPU occupancy. Moreover, threads which were assigned to pixels for which rasterization has been completed, cannot be reassigned to help other threads still working on active pixels. Because of this, SST can be seen as maintaining a good balance between memory and workload. In the following section, we evaluate SST and compare it to other methods.

6. Results and evaluation

We implemented our proposed rendering method in CUDA 5.0 which outputs a set of textures containing the normals and depth values of the rendered surface. As discussed above, except for

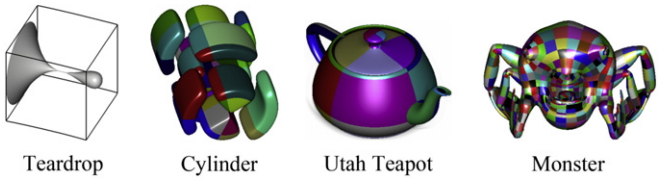

Fig. 7. Datasets used to benchmark the SST algorithm.

the CSG example in Fig. 8, we only consider the solution which is closest to the near plane. The data is then forwarded to GLSL performing Phong shading. All tests in this paper were run on a NVIDIA GeForce GTX 680 with a 4 GB memory, under Windows 7. The accompanying video shows screen captures of all examples shown in this paper.

6.1. Benchmarking results

Rendering implicit surfaces within deformed volumes and rendering parametric surfaces are different problems. Previous methods generally address one of these scenarios. A method designed for one of the two scenarios is generally difficult to extend so that it also performs well in the other scenario. However, by slightly modifying our framework, our algorithm can be used to also render parametric Bézier surfaces [22]: By omitting the third parameter direction, the Bézier volumes in the algorithm presented in Section 4 can be replaced with parametric Bézier surfaces. More details are given in Appendix B. Given this modification, we compare our method to Bézier Clipping [27]. Since a surface problem requires less memory than a volume problem, we are able to run Bézier Clipping exclusively through the (faster) local memory, where each pixel is executed in parallel.

Volume and surface datasets. By referring to the examples shown in Fig. 7, we test our method on two volume datasets, Teardrop and Cylinder, and compare rendering performance to a basic scheme, which maintains a subdivision tree for each screen pixel. Due to the high memory requirement of the basic scheme, the image resolution is limited to 512×512 for both datasets. Note that, the basic scheme is the GPU version of [21] following a similar strategy to render implicit surfaces.

We tested all three methods (SST, Bézier Clipping and basic scheme) on two parametric surface datasets, Teapot and Monster with a resolution of 512×512. While Bézier Clipping is more efficient than the basic scheme, it is still significantly slower than the presented SST method. The goal of this paper was not to implement an optimized framework of the modified Bézier clipping method on the improvement of numerical stability (e.g., [31]). Our work focus on the improvement of rendering speed. Note, for the surface case, the implementation of Bézier Clipping and the basic scheme is identical, except that Bézier Clipping uses a more efficient subdivision strategy, where the basic scheme splits patches in the middle. The method proposed in this paper is not only a faster alternative to these methods, but also numerically very robust. While tessellation methods such as [32] yield high efficiency in rendering parametric surfaces, it is not clear on how to extend them to the dynamic scenario where the surface properties change during the rendering.

CSG models. Constructive solid geometry (CSG) is often used in solid modeling and has a number of applications. We created complex models by using Boolean operators to combine simple volume datasets. A dice and a cheese are composed of simple spheres and cubes embedded in volumetric Bézier patches and rendered with a resolution of 512×512. We also deformed these two models which validate that our approach was able to preserve sharp edges, as shown in Fig. 8. It is challenging for polygonal meshes to preserve the exact sharp edges when surfaces are undergoing a deformation.

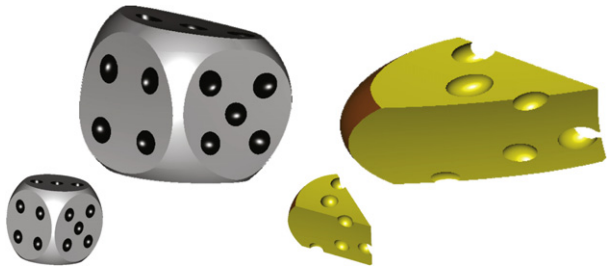

Fig. 8. Our algorithm is applied to deforming CSG models, where sharp edges are exactly preserved. This kind of application is challenging for polygonal representations.

Larger datasets. We run our algorithm on a wind blade dataset with 800 K Bézier patches with a resolution of 512×512. We vary the implicit surface during the visualization. Fig. 9 shows a few snapshots. Our approach is able to render this dataset size as it makes the economical use of GPU memory, by sharing subdivided among multiple pixels.

Timings. Table 1 gives an overview of the computation times for the experiments conducted above, and provides further relevant information. It can be seen that SST outperforms the basic scheme on all the six datasets. It is important to mention that SAG is excluded from the timings in Table 1, as it is applied to the other basic schemes and Bézier Clipping as well. Moreover, SAG is performed in parallel for those rays which overlap with the bounding boxes of the Bézier patches. Since for the given datasets, this requires much less computation time in comparison to the actual core algorithm, it can be neglected. In addition, we provide a breakdown of the timings for the different core stages (as illustrated in Fig. 10) of the SST algorithm.

6.2. Evaluation

Scalability of SST. Here we test how SST performs in terms of rendering speed and memory cost when the image resolution is increased. By referring to Fig. 11, image resolution is doubled at each step, where the computation time and memory consumption at most increases by a factor of four. This implies that by increasing the resolution, computation time and memory consumption only increase linearly. Note that, the corresponding plots are not provided for the latter two methods due to the limitation of local memory. However, the performance numbers in Table 1 indicate that SST significantly performs better than both, the basic scheme and Bézier Clipping, which both are implemented with CUDA on the GPU as well. While the memory consumption of the latter methods scale linearly as well, their performance degrade due to the excessive access of local memory.

In addition, we conduct a study on how SST scales with increasing degrees of the input patches. For this, we construct a tri-quadratic Bézier volume to represent the algebraic function $f(x, y, z) = x^2 + y^2 + z^2 - 1$ by using Marsden's identity [33]. We raise the degree of this initial tri-quadratic patch and construct a tri-cubic, tri-quartic, and tri-quintic patch. Note that each one of them exactly represents $f(x, y, z)$. In the study we visualize the implicit surface $f(x, y, z) = 0$, i.e., a sphere with radius 1. We run both, the SST algorithm and the basic scheme. Fig. 13 illustrates the growth in computation time by increasing degree. For the tri-quadratic case, both methods perform similarly. However, on the higher order patches SST outperforms the basic scheme. Since the basic scheme constructs a subdivision tree for each pixel, it also performs more subdivisions. As the subdivision cost is $O(p^3)$, where p is the degree, performing more subdivisions results in longer computation times.

Table 1
Row 1–4 show timings of SSTs and the basic scheme to render isosurfaces. Row 5 and 6 show timings of SSTs and Bézier clipping (B.C.) to render parametric surfaces. The table also shows the corresponding speedups by using SSTs.

	Screen occupancy (%)	# Patch	# Degree	SST (ms)	Basic (ms)	B.C. (ms)	Speedup
Teardrop	38.9%	1	5	75.8	3151.7	N/A	41.6
Cylinder	68.1%	50	3	151.2	1719.1	N/A	11.4
Dice	32.5%	28	2	136.0	1500	N/A	11.03
Cheese	32.5%	23	2	94.5	903.4	N/A	9.56
Teapot	68.2%	32	3	19.4	251.1	256.7	12.9/13.2
Monster	73.1%	1494	3	25.9	294.9	121.4	11.4/4.7

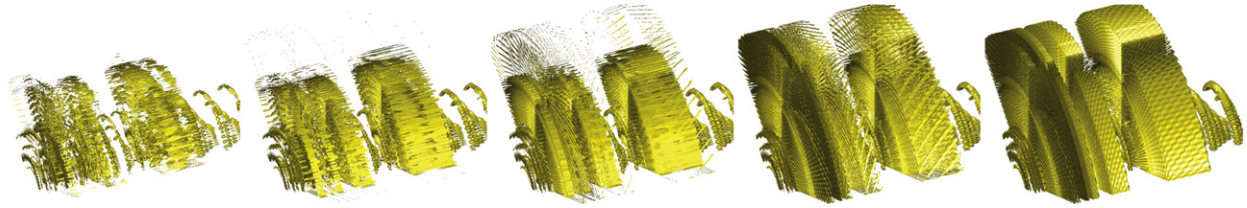

Fig. 9. Different isosurface snapshots rendering the velocity magnitude of air moving around a wind blade. The dataset consists of 800 K tri-quadratic rational Bézier patches (≈330 MB).

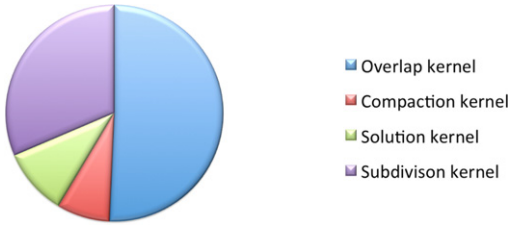

Fig. 10. The breakdown of the timings for the core of SST. Compute overlap kernel: 50.7%, Compaction kernel: 7.9%, Newton kernel: 10%, Subdivision kernel: 31.4%.

SST vs. pre-subdivision. A pre-subdivision stage is commonly employed to reduce the computation load per pixel: before rendering takes place, the volumetric representation is subdivided where subdivided volumes are only kept if they potentially contain a piece of the isosurface. For a parametric surface, all subdivided patches are kept. In the following we compare SST to the basic scheme and Bézier Clipping by adding several pre-subdivision steps. As illustrated in Fig. 12, it can be seen that the basic scheme and Bézier Clipping converge towards our algorithm as the number of pre-subdivisions increases.

While a pre-subdivision stage can help to significantly improve the rendering time, it is generally not clear how many initial subdivision steps should be performed. Generally, there might be more pre-subdivisions than necessary, i.e., for many rays, Newton's method would already succeed on the coarser or even initial patch, or, the number of pre-subdivisions is not enough, i.e., the number of pre-subdivisions are too few to have an impact on the number of local subdivisions and hence on the rendering speed. In SST, more subdivision work is only performed for more complex areas (e.g., around the silhouette) where Newton's method requires more accuracy to succeed. In particular, for larger datasets (Fig. 9), SST has a lower memory footprint compared to employing a pre-subdivision stage. Next to those reasons, the main reason for this is that SST considers only those parts of the dataset which are potentially visible from the camera. An additional benefit of this is a more efficient rendering speed.

7. Conclusions

In this paper, we propose *Shared Subdivision Trees* which efficiently map to the GPU by sharing a single subdivision tree for a given patch across multiple pixels. We demonstrate that Shared Subdivision Trees cannot only be used to efficiently rasterize implicit surfaces embedded in deforming volumes but also to rasterize parametric surfaces undergoing a deformation. Our method, based on a direct rendering approach, is especially useful in applications such as free-form deformation or physically-based animation applied to solids, where methods based on meshing, tessellation, or sampling, are difficult to apply due to the nonlinear deformation of the given surface representation. Input is allowed to be rational, allowing the method to also be used in other applications such as CAD modeling or within isogeometric analysis. The presented algorithm is simple in terms of implementation and executes efficiently on GPU. In the future we plan to embed this algorithm in time critical applications such as 3D animation to render nonlinear effects more robustly and efficiently. We further want to explore our scheme to work directly with more general volumetric representations, such as trivariate NURBS, without converting them first into a Bézier representation.

Acknowledgments

This research, which is carried out at BeingThere Centre, is supported by the Singapore National Research Foundation under its International Research Centre @ Singapore Funding Initiative and administered by the IDM Programme Office. This research is partially supported by Multi-platform Game Innovation Centre (MAGIC) GREaT IPMD13013, funded by the Singapore National Research Foundation under its IDM Futures Funding Initiative and administered by the Interactive & Digital Media Programme Office, Media Development Authority. Sai-Kit Yeung is supported by Singapore University of Technology and Design (SUTD) StartUp Grant ISTD 2011 016, SUTD-ZJU Collaboration Research Grant 2012 SUTD-ZJU/RES/03/2012, SUTD-MIT International Design Centre Grant IDG31300106 and Singapore MOE Academic Research Fund MOE2013-T2-1-159. Fuchang Liu is partially supported by NSFC (61332017).

Appendix A. Pseudocode of SST algorithm

Inputs are M screen pixels, and a list $\mathcal{L}$ of patches in perspective space. Output is a solution vector $\mathcal{R}$, which stores a solution for each pixel. As discussed in Section 4, line 3 corresponds to stage 1, line 4–10 corresponds to stage 2, and line 11–15 corresponds to stage 3. The subdivision operation in line 13 splits patch $\mathcal{L}_i$ at the parametric center into eight Bézier patches and replaces patch $\mathcal{L}_i$ in $\mathcal{L}$ with the respective subpatches.

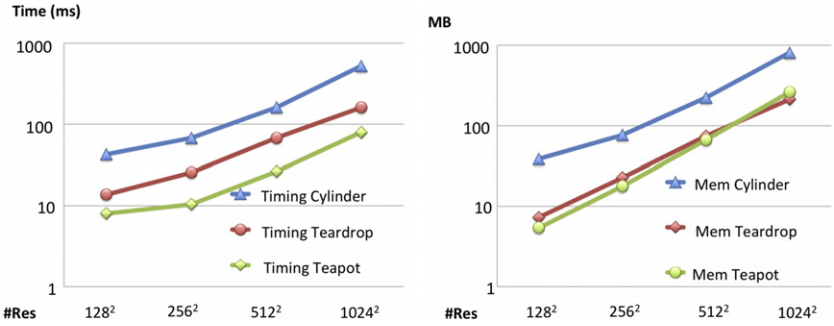

Fig. 11. Performance (left), and memory cost of three volume datasets (right), with respect to the image resolution.

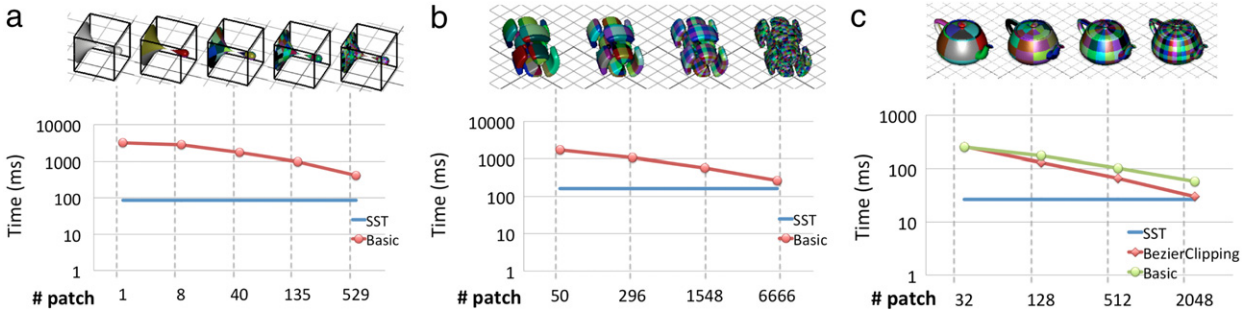

Fig. 12. The performance with respect to the number of patches for (a) teardrop, (b) cylinder, and (c) teapot.

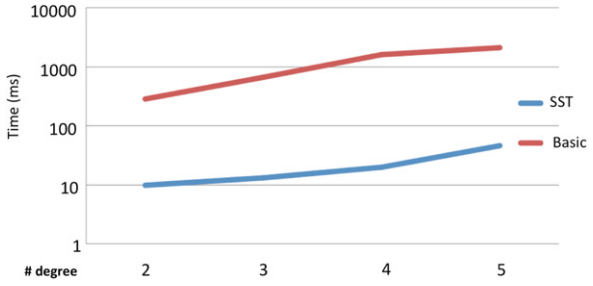

Fig. 13. The performance with respect to the order of patches.

1: Initialize 1D array $\mathcal{R} \leftarrow \{\infty\}_{k=1}^{M}$
2: **do**
3: $\mathcal{I} \leftarrow ComputePixelOverlap(\mathcal{L})$ **in parallel**
4: **for** each tuple $(s, t, i) \in \mathcal{I}$ **in parallel do**
5: $\mathbf{o}_{st} \leftarrow \{b_s, b_t, \hat{a}\}$
6: **if** $\mathbf{r}_{st} \leftarrow Newton(\mathcal{L}_i, \mathbf{o}_{st})$ succeeds **then**
7: $\mathcal{R}_{st} \leftarrow \min(\mathcal{R}_{st}, \mathbf{r}_{st})$
8: Remove (s, t, i) from $\mathcal{I}$
9: **end if**
10: **end for**
11: $\mathcal{F} \leftarrow \{\}$
12: **for** each tuple $(s, t, i) \in \mathcal{I}$ **in parallel do**
13: $\mathcal{L} \leftarrow \{Subdivide(\mathcal{L}_i)$ if $i \notin \mathcal{F}\}$
14: $\mathcal{F} \leftarrow \mathcal{F} \cup \{i\}$
15: **end for**
16: **while** $\mathcal{I} \neq \emptyset$

Appendix B. Rasterization of parametric surfaces

Section 4 describes an algorithm to rasterize an isosurface against a grid of pixels. The core of the algorithm is the formulation of three constraint equations, where the first equation requires a solution to lie sufficiently close to the isosurface. The formulation of the rasterization problem of a (rational) parametric surface in perspective space, is defined as

$$\mathbf{S}(u, v) = \frac{\sum_{i=0}^{n}\sum_{j=0}^{m} w_{ij}\,\mathbf{c}_{ij}\,\theta_i^n(u)\,\theta_j^m(v)}{\sum_{i=0}^{n}\sum_{j=0}^{m} w_{ij}\,\theta_i^n(u)\,\theta_j^m(v)} = \{x(u, v), y(u, v), z(u, v)\}.$$

Given pixel (s, t), the goal is to compute all points which lie on the parametric surface $\mathbf{S}$, and within pixel (s, t). The two scalars, b_s and b_t, defined as above, are used to define two perpendicular planes, $x - b_s = 0$ and $y - b_t = 0$, coinciding with the center of the pixel. Substituting $\mathbf{S}(u, v)$ into the two plane equations yields the two constraint inequalities, $|x(u, v) - b_s| \leq p_w$, and $|y(u, v) - b_t| \leq p_w$, where p_w is the pixel width as defined above.

Given these two inequalities, a new 2D Bézier surface with control points $\{x_{ijk} - b_s, y_{ijk} - b_t\}$ is constructed and input to the algorithm as described in Section 4 with slight modifications: (1) patches are only subdivided along u and v which is responsible for the significant speedup and reduced memory requirement in contrast to the isosurface rasterization; (2) Newton's method is conducted in 2D instead in 3D, as the parametric surface is a function of two variables u and v, respectively. Pixel depth is determined by evaluating $z(u, v)$.

References

[1] Sederberg TW, Parry SR. Free-form deformation of solid geometric models. In: SIGGRAPH. New York: ACM Press; 1986. p. 151–60.
[2] Hughes TJ, Cottrell JA, Bazilevs Y. Isogeometric analysis: Cad, finite elements, nurbs, exact geometry, and mesh refinement. Comput Methods Appl Mech Engrg 2005;194:4135–95.
[3] Barr AH. Ray tracing deformed surfaces. In: SIGGRAPH. New York: ACM Press; 1986. p. 287–96.
[4] Rockwood A. Accurate display of tensor product isosurfaces. In: IEEE VIS. San Francisco: IEEE Press; 1990. p. 353–60.
[5] Reimers M, Seland J. Ray casting algebraic surfaces using the frustum form. Comput Graph Forum 2008;27(2):361–70.
[6] Knoll A, Hijazi Y, Hansen CD, Wald I, Hagen H. Interactive ray tracing of arbitrary implicit functions. In: IEEE IRT. UIm: IEEE Press; 2007. p. 11–8.

[7] Parker S, Shirley P, Livnat Y, Hansen C, Sloan PP. Interactive ray tracing for isosurface rendering. In: IEEE VIS. Los Alamitos: IEEE Press; 1998. p. 233–8.
[8] Entezari A, Dyer R, Moller T. Linear and cubic box splines for the body centered cubic lattice. In: IEEE VIS. Washington: IEEE Press; 2004. p. 11–8.
[9] Kim M, Entezari A, Peters J. Box spline reconstruction on the face-centered cubic lattice. IEEE Trans Vis Comput Graphics 2008;14(6):1523–30.
[10] Liu B, Clapworthy GJ, Dong F. Fast isosurface rendering on a gpu by cell rasterization. Comput Graph Forum 2009;28(8):2151–64.
[11] Chang YK, Rockwood A, He Q. Direct rendering of freeform volumes. Comput-Aided Des 1995;27(7):553–8.
[12] Meyer M, Nelson B, Kirby R, Whitaker R. Particle systems for efficient and accurate high-order finite element visualization. IEEE Trans Vis Comput Graphics 2007;13(5):1015–26.
[13] Zwicker M, Pauly M, Knoll O, Gross M. Pointshop 3D: an interactive system for point-based surface editing. ACM Trans Graph 2002;21(3):322–9. (Proceedings SIGGRAPH 2002, San Antonio).
[14] Lorensen WE, Cline HE. Marching Cubes: A high resolution 3D surface construction algorithm. In: SIGGRAPH. New York: ACM Press; 1987. p. 163–9.
[15] Cignoni P, Floriani LD, Montani C, Puppo E, Scopigno R. Multiresolution modeling and visualization of volume data based on simplicial complexes. In: VVS, New York. 1994. p. 19–26.
[16] Nelson B, Kirby RM. Ray-tracing polymorphic multidomain spectral/hp elements for isosurface rendering. IEEE Trans Vis Comput Graphics 2006; 12(1):114–25.
[17] Nelson B, Liu E, Kirby R, Haimes R. Elvis: a system for the accurate and interactive visualization of high-order finite element solutions. IEEE Trans Vis Comput Graphics 2012;18(12):2325–34.
[18] Uffinger M, Frey S, Ertl T. Interactive high-quality visualization of higher-order finite elements. Comput Graph Forum 2010;29(2):337–46.
[19] Muller M, Gross M. Interactive virtual materials. In: GI. Waterloo. 2004. p. 239–46.
[20] Elber G, Kim MS. Geometric constraint solver using multivariate rational spline functions. In: SMA. New York: ACM Press; 2001. p. 1–10.
[21] Martin T, Cohen E, Kirby MM. Direct isosurface visualization of hex-based high-order geometry and attribute representations. IEEE Trans Vis Comput Graphics 2012;18(5):753–66.
[22] Farin G. Curves and surfaces for CAGD: a practical guide. 5th ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc.; 2002.
[23] Kajiya JT. Ray tracing parametric patches. In: SIGGRAPH. New York: ACM Press; 1982. p. 245–54.
[24] Cohen E, Riesenfeld RF, Elber G. Geometric modeling with splines: an introduction. Natick, MA, USA: A. K. Peters, Ltd.; 2001.
[25] Parker S, Martin W, Sloan PJ, Shirley P, Smits B, Hansen C. Interactive ray tracing. In: I3D. New York: ACM Press; 1999. p. 119–26.
[26] Toth DL. On ray tracing parametric surfaces. In: SIGGRAPH. New York: ACM Press; 1985. p. 171–9.
[27] Nishita T, Sederberg TW, Kakimoto M. Ray tracing trimmed rational surface patches. In: SIGGRAPH. New York: ACM Press; 1990. p. 337–45.
[28] NVIDIA: CUDA programming guide 2.0, 2008. http://developer.nvidia.com/object/cuda.html.
[29] Hoberock J, Bell N. Thrust: a parallel template library, 2010. URL: http://thrust.github.io/.
[30] Zhou K, Hou Q, Wang R, Guo B. Real-time kd-tree construction on graphics hardware. ACM Trans Graph 2008;27(5): Article No. 126. (Proceedings SIGGRAPH Asia 2008, Singapore).
[31] Efremov A, Havran V, Seidel HP. Robust and numerically stable bezier clipping method for ray tracing nurbs surfaces. In: SCCG. New York: ACM Press; 2005. p. 127–35.
[32] Eisenacher C, Meyer Q, Loop C. Real-time view-dependent rendering of parametric surfaces. In: I3D. New York: ACM Press; 2009. p. 137–43.
[33] Marsden MJ. An identity for spline functions with applications to variation diminishing spline approximation. J Approx Theory 1970;3(7):7–49.