Performing Efficient NURBS Modeling Operations on the GPU

Adarsh Krishnamurthy, Rahul Khardekar, Sara McMains, Kirk Haller, and Gershon Elber

Abstract—We present algorithms for evaluating and performing modeling operations on NURBS surfaces using the programmable fragment processor on the Graphics Processing Unit (GPU). We extend our GPU-based NURBS evaluator that evaluates NURBS surfaces to compute exact normals for either standard or rational B-spline surfaces for use in rendering and geometric modeling. We build on these calculations in our new GPU algorithms to perform standard modeling operations such as inverse evaluations, ray intersections, and surface-surface intersections on the GPU. Our modeling algorithms run in real time, enabling the user to sketch on the actual surface to create new features. In addition, the designer can edit the surface by interactively trimming it without the need for retessellation. Our GPU-accelerated algorithm to perform surface-surface intersection operations with NURBS surfaces can output intersection curves in the model space as well as in the parametric spaces of both the intersecting surfaces at interactive rates. We also extend our surface-surface intersection algorithm to evaluate self-intersections in NURBS surfaces.

Index Terms—NURBS, GPU, inverse evaluation, sketching, interactive trimming, SSI, intersection curves, self-intersection, prefix sum.

1 Introduction

Industrial design of products has shifted from using boxy shapes with straight edges to incorporate curved free-form surfaces. Non Uniform Rational B-Spline (NURBS) surfaces provide a convenient and compact representation of such curved surfaces that has become the representation of choice in mechanical CAD systems. Hence, real-time interaction with NURBS surfaces is essential for any CAD package. However, since evaluation of a NURBS surface is inherently a computation-intensive process, commercial CAD packages deal with it by preprocessing NURBS surfaces, usually tessellating them and using the triangulated model for display as well as certain modeling operations like selection. With the advent of programmable graphics hardware, the need for tessellating the NURBS surface in the CPU for display was obviated, since the GPU can be used for the evaluation and direct display of the surfaces [1], [2], [3], [4]. However, CAD packages still perform modeling operations using the CPU with either the tessellated surfaces or analytically using NURBS definitions. This reduces the interactivity for the user when designing these free-form surfaces, since operations like sketching on the NURBS surface or fast evaluation of intersection curves are not possible. Leading commercial CAD packages do not allow the designer to sketch directly on the NURBS surface; instead, they restrict the user to sketching on a tangent plane. Because of this, the designer has to wait until the operation is completed to get visual feedback.

The process of finding the surface coordinates (x, y, z) for given parameter values (u, v) is called evaluation. Inverse evaluation is the process of finding the parameter values (u, v) given any point on the surface. We have developed a parallel algorithm for fast inverse evaluations of NURBS surfaces on the GPU. This algorithm forms the basis of many modeling operations like selection (ray-surface intersection), sketching on the surface, and interactive trimming (see Fig. 1). Moreover, since these algorithms exploit the parallelism of the GPU, these operations can now be performed at interactive speeds, making immediate visual feedback to the designer possible for the first time. We demonstrate the use of our fast inverse evaluation algorithm to directly sketch on the surface, which makes certain operations like interactive trimming intuitive to the designer.

Designers are usually trained to work with curves on surfaces, such as silhouette curves and intersection curves. Thus, they would like to see real-time changes in these curves as the underlying surfaces are edited, which requires an efficient algorithm to compute intersection curves of free-form surfaces. Finding the intersection curve is, in general, a very complex operation, since two NURBS surface equations of arbitrary degree have to be solved simultaneously. Many commercial CAD packages use marching methods, where the algorithm uses a numerical root-finding technique to first find a single intersection point. The algorithm then finds another point along the intersection curve that is close to the first intersection point. This process is repeated and ultimately a complete piecewise linear approximation of the intersection curve is calculated. However, since this technique is inherently serial, it cannot be parallelized for efficient evaluation on the GPU. We have developed a GPU-accelerated parallel algorithm to evaluate the intersection curves using bounds on the evaluated surface points. This algorithm is both fast and guaranteed to find the intersection curves within a user-defined tolerance.

- *A. Krishnamurthy, R. Khardekar, and S. McMains are with the Department of Mechanical Engineering, University of California, Berkeley, Berkeley, CA 94720. E-mail: {adarsh, rahul, mcmains}@me.berkeley.edu.*
- *K. Haller is with SolidWorks Corporation, 300 Baker Avenue, Concord, MA 01742. E-mail: khaller@solidworks.com.*
- *G. Elber is with the Department of Computer Science, Technion, Israel Institute of Technology, Haifa 32000, Israel. E-mail: gershon@cs.technion.ac.il.*

Manuscript received 24 July 2008; revised 21 Nov. 2008; accepted 7 Jan. 2009; published online 5 Feb. 2009.
Recommended for acceptance by H. Qin.
For information on obtaining reprints of this article, please send e-mail to: tvcg@computer.org, and reference IEEECS Log Number TVCGSI-2008-07-0107.
Digital Object Identifier no. 10.1109/TVCG.2009.29.

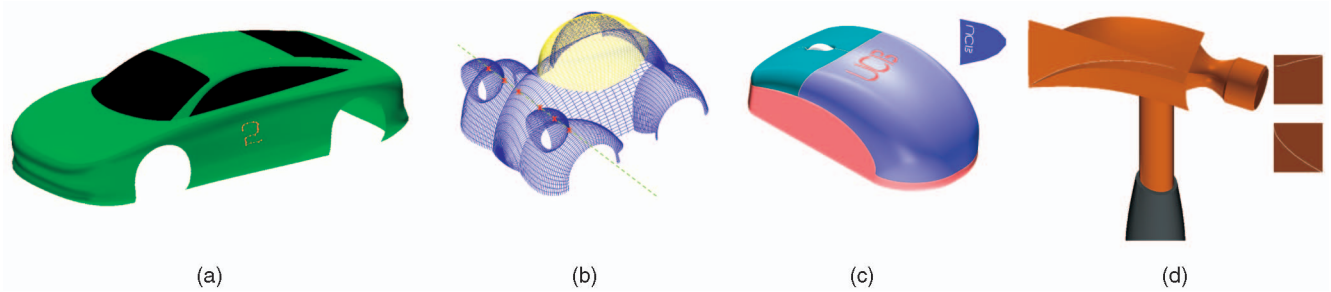


Fig. 1. Modeling operations like (a) sketching, (b) ray intersection, (c) trimming, and (d) surface-surface intersection performed directly on trimmed NURBS models.

In this paper, we present GPU-based algorithms to perform modeling operations efficiently on NURBS surfaces. Our main contributions include:

- A new unified method to calculate exact derivatives and exact normals of arbitrary-degree NURBS surfaces on the GPU. Our method is designed so as to not require separate fragment programs for evaluating surfaces of different degrees.
- An efficient algorithm to perform inverse evaluation of NURBS surfaces on the GPU. This algorithm finds the parametric (u, v) coordinate given any (x, y, z) coordinate on the NURBS surface within an arbitrary user-defined tolerance.
- A novel method to interactively trim and sketch on a NURBS surface in real time. This is possible because our fast inverse evaluation algorithm enables us to sketch in the model space, not just in the parametric space, with the correspondence tracked simultaneously.
- A GPU-accelerated algorithm to perform fast and robust NURBS surface-surface intersections. The intersection curve, like the sketch curve above, is simultaneously output in the model space as well as in the parametric spaces of the two NURBS surfaces. The GPU is used to accelerate the operation by finding points on the intersection curves and the actual intersection curves are calculated from these points on the CPU.
- An extension of the surface-surface intersection algorithm to evaluate self-intersections in NURBS surfaces. This algorithm can be used to detect self-intersections and output the intersection curves if the surface is self intersecting.

We summarize our approach to evaluating and rendering NURBS surfaces on the GPU in Section 2; for more details, refer to [1]. We then discuss the evaluation of first and second derivatives of the NURBS surfaces (Section 3) and then use these to compute bounding boxes for NURBS surfaces (Section 4). Then, we describe how these bounds are used to perform inverse evaluations (Section 5) and compute intersection curves (Section 6). Fig. 2 shows these connections between the different parts of our algorithms; each of these operations are described in detail in the sections indicated.

2 Previous Work

One of the main prerequisites for performing fast modeling operations on NURBS is to have a fast NURBS evaluator. We present a short outline of our algorithm to evaluate NURBS surfaces on the GPU that was explained in detail in [1]. The main idea of our algorithm was to use a fragment program to evaluate a NURBS surface in several passes. One advantage of our approach is that we have two corresponding representations of the NURBS surface as four-component vectors—(x, y, z, w) coordinates—in space as well as their corresponding parametric values—(u, v) coordinates. We exploit this correspondence during modeling operations like inverse evaluation and evaluation of intersection curves.

In our algorithm, we first evaluate the basis function values on the GPU in parallel for all the parameter positions where we want to evaluate the surface coordinates. The parameter positions can be chosen arbitrarily by the user; we chose an equally spaced grid of points to make the implementation simpler. For example in this paper, we chose a starting grid size of $1,024 \times 1,024$ and refined it based on the user-specified tolerances. We parallelize the de Boor evaluation algorithm [5] so that it runs efficiently on the GPU. We use the de Boor evaluation method because B-spline basis functions of any degree can be evaluated using the same fragment program. Other NURBS evaluations on the GPU either require different fragment programs for different degrees [2] or are restricted to cubic polynomials [3]. We perform the basis function evaluation separately for the u and v parametric directions on the GPU and store these values as textures. We multiply these basis function values with the corresponding control points to obtain the surface point

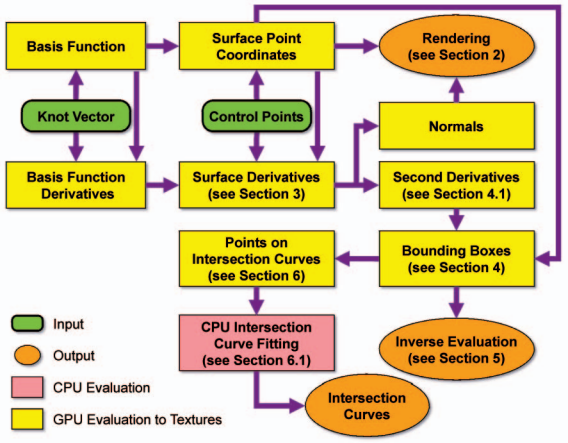


Fig. 2. Graphic showing the links between different parts of our modeling algorithms. The results of the GPU evaluations are stored in separate textures.

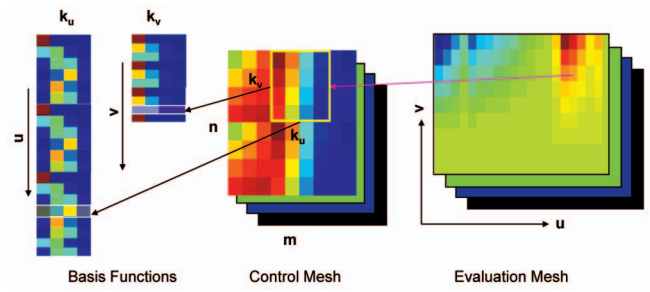


Fig. 3. Graphic showing our NURBS evaluation algorithm on the GPU. The control mesh of size $m \times n$ and the evaluation mesh are made of four-component vectors stored as RGBA textures. The surface patch is of order k_u in the u direction and k_v in the v direction. The multiplication is restricted to the submesh of size $k_u \times k_v$.

coordinates. We can restrict the multiplication to the submesh of the control points that affect a particular surface point because of the local support property of NURBS (Fig. 3). We perform the multiplication operation in parallel for all the surface coordinates using another fragment program. The surface point coordinates thus calculated are stored directly in a texture on the GPU using the RGBA channels. While rendering, we interpret these values stored in the texture as vertex coordinates using a Vertex Buffer Object (VBO). We thus avoid the slow operation of reading back the computed data from the GPU to the CPU, directly rendering the NURBS surface on the screen.

Previous work that used GPUs to render NURBS curves or surfaces focused only on efficient evaluation of the surface coordinates and/or normals by the authors of [2], [3], [6], [7]. They did not use GPUs to perform modeling operations like inverse evaluations and intersection curve evaluations. Previous work on inverse evaluation of NURBS surfaces mainly focused on ray tracing NURBS surfaces. Ray tracing was performed on parametric and rational surfaces by solving for the ray-surface intersection point using numerical methods [8], [9], [10]. There has also been previous work on ray tracing using the GPU, which includes [4], [11], [12], [13]. Another application of inverse evaluation of NURBS is solving for geometric constraints. A method to solve geometric constraints by using multivariate splines was given in [14], which can be used to solve several related problems like ray traps and sweep envelopes. Inverse evaluation has also been used for haptic rendering to find the parametric (u, v) coordinates of a given point on a NURBS surface [15]. Inverse evaluation was used in this case to solve for the contact point of a haptic probe with trimmed NURBS surfaces in a virtual environment.

Carr et al. [13] also presented a GPU algorithm to find the indexes of the rendered texels in a texture, a subproblem for our GPU algorithm for intersection calculations. This subproblem falls under the class of stream reduction, the process of removing unwanted elements from a stream of values and reducing it to a smaller list containing the required output. General Purpose computing on the GPU (GPGPU) uses stream reduction to remove defunct elements from the output of a previous pass before sending it as input for the next pass. Since the positions of the output elements do not have any fixed correspondence with the positions of the input, the stream reduction process is considered nonuniform. A parallel $O(k + \log n)$ algorithm, where k is the output size, for nonuniform stream reduction based on prefix sums was given in [16]. However, standard graphics cards do not have the capability to perform a scatter operation (random writes to different memory locations), which was an essential step in the algorithm given in [16]. Another algorithm has been presented in [17] for nonuniform stream reduction on the GPU that runs in $O(n \log n)$, not as efficient due to workarounds required because of lack of scatter. A stream reduction algorithm specifically for 2D textures on the GPU was proposed in [18], which used the fragment processor to perform other operations while performing the scatter operation, thereby hiding the latency. Recently, an $O(n)$ GPU stream reduction algorithm was proposed in [19], also using prefix sums, that relies on the latest NVIDIA CUDA architecture for its scatter functionality. We propose a similar $O(n)$ stream reduction algorithm based on computing a parallel prefix sum, but implement it using the standard GPGPU framework so that it is both compatible with older hardware and not limited to a single brand of GPU.

Several approaches to collision detection on the GPU have been proposed. Occlusion queries on graphics hardware were used in [20] to detect collisions of polygonal meshes in large environments. Collisions between particles were calculated in [21], [22] to simulate large scale particle systems on the GPU. Recently, a method to detect collisions between deformable parameterized surfaces using GPUs was presented by Greß et al. [18]. They solve the collision detection problem by generating a bounding box hierarchy for the surface and then detecting collisions by checking overlap between the bounding boxes.

Evaluation of intersection curves is a fundamental operation in computer-aided geometric design and solid modeling [23], [24]. There have been several attempts to solve the problem, since it is hard to achieve all the desired characteristics of robustness, accuracy, and efficiency. A comprehensive survey of surface-surface intersection algorithms was summarized in [25]. A more recent algebraic algorithm for efficient surface intersection using lower dimensional formulations was given by Krishnan and Manocha [26]. They also classified the conventional methods for evaluating the intersection curves as analytical methods, lattice evaluations, subdivision methods, and marching methods. Many commercial CAD software packages use the numerical marching method outlined in [27], [28] to evaluate intersection curves.

3 Derivatives of NURBS Surfaces

To perform geometric operations on NURBS surfaces, we not only require the surface point coordinates themselves but also the first and second partial derivatives with respect to the two parameter directions u and v at the surface points. As a very fast first-degree approximation, we can use the evaluated point coordinates to estimate the first derivatives using central differencing. However, this approach gives rise to artificial discontinuities at patch boundaries and at rational parts of the surface. Moreover, second derivatives estimated from these first derivatives in the same manner have larger errors associated with them. One way to overcome this issue is to evaluate the normals

of the surface exactly at each surface point, similar to the evaluation of the surface coordinates. Since we already evaluate the higher order basis functions from lower order basis-functions, we can directly calculate the derivatives of the basis functions within the same framework as our basis function evaluation algorithm, and then use the basis function derivatives to evaluate the derivatives of the NURBS surface precisely, to within machine precision.

3.1 Differential Geometry for B-Spline Surfaces

In this section, we present a concise version of the equations that are required for computing derivatives of NURBS surfaces, adapted from [29]. We present the exact equations for a Non-Uniform B-Spline (NUBS) surface first and then extend the derivation to include rational surfaces. For a NUBS surface, $S(u,v)$, given by (1), the derivatives can be computed by multiplying the control points (P_{ij}s) with the derivatives of the basis functions. The variables N_i^ps and N_j^qs are the B-spline basis functions of degrees p and q, respectively, as a function of the knots u_is and v_is, respectively ((2) and (3)); the P_{ij}s are the NUBS control points defined as a quadrilateral mesh.

$$S(u,v) = \sum_{i=0}^{n}\sum_{j=0}^{m} N_i^p(u) N_j^q(v) P_{ij}, \tag{1}$$

$$N_i^p(u) = \frac{u-u_i}{u_{i+p}-u_i} N_i^{p-1}(u) + \frac{u_{i+p+1}-u}{u_{i+p+1}-u_{i+1}} N_{i+1}^{p-1}(u), \tag{2}$$

$$N_i^0(u) = \begin{cases} 1, & \text{if } u_i \le u < u_{i+1}, \\ 0, & \text{otherwise.} \end{cases} \tag{3}$$

The derivative of the basis function of degree p with respect to u is given by (4). To evaluate the derivative of a basis function of degree p, the basis function of degree $p-1$ needs to be computed. We use the indicial notation $N_{,u}$ to denote the derivative with respect to u. Note that $p-1$ in the numerator of (4) arises due to the fact that the B-spline basis function of degree p that we are differentiating is a piecewise polynomial of degree p in u.

$$N_{i,u}^p(u) = \frac{p-1}{u_{i+p}-u_i} N_i^{p-1}(u) - \frac{p-1}{u_{i+p+1}-u_{i+1}} N_{i+1}^{p-1}(u). \tag{4}$$

The derivatives of the B-spline basis functions, $N_{,u}$ and $N_{,v}$, are then multiplied by the control points P_{ij} to get the derivative along the u or v parametric direction on the surface as given in (5) and (6), respectively. We can then calculate the surface normal $N(u,v)$ of the NUBS surface (Fig. 4) by taking the cross product of the u and v partial derivatives (7). It should be noted that $N(u,v)$ is not a unit vector field but it is well defined as long as S is a regular surface.

$$S_{,u}(u,v) = \sum_{i=0}^{n}\sum_{j=0}^{m} N_{i,u}^p(u) N_j^q(v) P_{ij}, \tag{5}$$

$$S_{,v}(u,v) = \sum_{i=0}^{n}\sum_{j=0}^{m} N_i^p(u) N_{j,v}^q(v) P_{ij}, \tag{6}$$

$$N(u,v) = S_{,u}(u,v) \times S_{,v}(u,v). \tag{7}$$

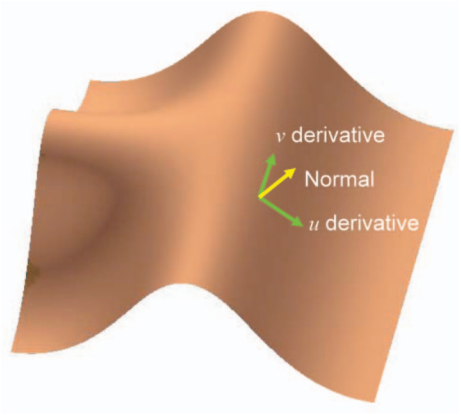


Fig. 4. Calculation of surface normal from the u and v partial derivatives.

3.2 Rational Derivatives

The derivatives of NURBS surfaces are not as straightforward to evaluate as in the NUBS case [30]. This is because the derivatives have to be evaluated using the chain rule due of the existence of the rational component. The NURBS surface coordinates are evaluated as the four-component vector shown in (8) and (9). Since we evaluate the four-component vectors without performing the rational division on the GPU, we can effectively use this data to evaluate the surface derivatives.

$$S(u,v) = \frac{\bar{X}}{w}, \quad \bar{X} = \begin{pmatrix} x \\ y \\ z \end{pmatrix}, \tag{8}$$

$$\begin{pmatrix} x \\ y \\ z \\ w \end{pmatrix} = \begin{pmatrix} \sum_{i=0}^{n}\sum_{j=0}^{m} N_i^p(u) N_j^q(v) x_{ij} \\ \sum_{i=0}^{n}\sum_{j=0}^{m} N_i^p(u) N_j^q(v) y_{ij} \\ \sum_{i=0}^{n}\sum_{j=0}^{m} N_i^p(u) N_j^q(v) z_{ij} \\ \sum_{i=0}^{n}\sum_{j=0}^{m} N_i^p(u) N_j^q(v) w_{ij} \end{pmatrix}, \tag{9}$$

$$S_{,u}(u,v) = \frac{\bar{X}_{,u} w - \bar{X} w_{,u}}{w^2}, \tag{10}$$

$$\begin{pmatrix} x_{,u} \\ y_{,u} \\ z_{,u} \\ w_{,u} \end{pmatrix} = \begin{pmatrix} \sum_{i=0}^{n}\sum_{j=0}^{m} N_{i,u}^p(u) N_j^q(v) x_{ij} \\ \sum_{i=0}^{n}\sum_{j=0}^{m} N_{i,u}^p(u) N_j^q(v) y_{ij} \\ \sum_{i=0}^{n}\sum_{j=0}^{m} N_{i,u}^p(u) N_j^q(v) z_{ij} \\ \sum_{i=0}^{n}\sum_{j=0}^{m} N_{i,u}^p(u) N_j^q(v) w_{ij} \end{pmatrix}. \tag{11}$$

The partial derivative with respect to u (10) is derived using the quotient rule (in turn derived using the chain rule). It can be calculated by first evaluating the product of the derivatives of the basis functions and the corresponding control points as a four-component vector (11), and then performing the required rational division operations. The partial derivative of the surface with respect to v can also be evaluated in a similar manner. In this work, we assume all the weights (w) are positive, and hence, no poles can occur in S or its partial derivatives.

3.3 GPU Implementation

The GPU implementation of the evaluation of surface derivatives is a direct extension of the evaluation of the surface coordinates as explained in Section 2. The GPU

evaluation consists of four steps as given below. The first three steps are similar to the method for evaluation of the surface coordinates. We give the steps for evaluating the surface derivatives with respect to u; the steps for finding the derivative with respect to v are similar, exchanging u and v in step 2:

1. Locate the submesh of control points that influence the evaluation point coordinates.
2. Compute the basis functions and the derivatives of the basis functions along the two-parameter directions, respectively.
   a. Compute the nonzero basis function derivatives with respect to u.
   b. Compute the nonzero basis functions with respect to v.
3. Multiply the nonzero basis functions and the basis function derivatives with their corresponding control points from the submesh and sum the results.
4. Evaluate the rational derivatives as given in (10) using the evaluated surface coordinates and surface derivatives from the previous step.

One notable feature of this algorithm is that step 1 and step 2b are already performed while evaluating the surface coordinates using our NURBS evaluation algorithm. Moreover, computing the u derivative in step 2a is different from evaluating the B-spline basis function only in the final step of the evaluation. Since we are using the de Boor evaluation algorithm, evaluating the B-spline basis function of order k as well as its derivative requires the evaluation of the B-spline basis function of order $k-1$. In practice, since we are already computing the B-spline basis function of order $k-1$, we store this intermediate result as a texture on the GPU. We then use this as input for evaluating both the B-spline basis function of order k as well as its derivative.

We evaluate the derivatives of the basis functions with respect to each parameter direction separately and store them in separate textures on the GPU. Once the derivatives with respect to the u and v directions are calculated as four-component vectors, the surface normals are calculated. This is performed using a separate fragment program that takes the rational surface derivatives as input and then evaluates their cross product to calculate the surface normal (7). Thus, the process of evaluating the NURBS surfaces as well as their normals can be performed efficiently within a single framework using our method.

4 Bounding Boxes for NURBS Surfaces

We make use of axis-aligned bounding boxes (AABB) for the NURBS surfaces to perform modeling operations using the GPU. With the help of such bounding boxes, several queries such as ray-surface intersections and surface-surface intersections can be efficiently answered, which then form the building blocks for more complex operations like sketching on the surface and intersection curve calculations. There are different methods to construct bounding boxes for free-form surfaces. One method is to fit bounding boxes that enclose the control points that define the surface. This method, however, does not produce very tight bounding boxes and makes the bounding boxes independent of the user-defined tolerance values. Another approximate method is to construct bounding boxes enclosing sets of four adjacent points evaluated on the surface. In [18], the bounding boxes for use in collision detection were constructed from sets of four adjacent points on a parameterized surface, after ensuring that their approximation of the surface is within the given tolerance by very finely subdividing the surface. However, this method does not guarantee that the surface will be completely enclosed by the bounding box and it can potentially miss some intersections. We overcome these issues by evaluating the NURBS surface in a regular grid and then expand the bounding boxes based on the curvature of the surface so that they are guaranteed to enclose the surface. Another advantage of this method is that the bounding boxes automatically become tighter when we evaluate the surface at a finer resolution.

The analytical expression for the factor that can be used to expand the bounding boxes based on the surface curvature is given by Filip et al. [31]. They show that if a parametric C^2 surface is evaluated at $(n+1)\times(m+1)$ grid of points, the deviation of the surface from the piecewise linear approximation cannot exceed a constant K defined by (12)-(15):

$$K = \frac{1}{8}\left(\frac{1}{n^2}M_1 + \frac{2}{nm}M_2 + \frac{1}{m^2}M_3\right), \tag{12}$$

$$M_1 = \max_{\forall(u,v)}\left[\max\left(\left|\frac{\partial^2 x}{\partial u^2}\right|, \left|\frac{\partial^2 y}{\partial u^2}\right|, \left|\frac{\partial^2 z}{\partial u^2}\right|\right)\right], \tag{13}$$

$$M_2 = \max_{\forall(u,v)}\left[\max\left(\left|\frac{\partial^2 x}{\partial u \partial v}\right|, \left|\frac{\partial^2 y}{\partial u \partial v}\right|, \left|\frac{\partial^2 z}{\partial u \partial v}\right|\right)\right], \tag{14}$$

$$M_3 = \max_{\forall(u,v)}\left[\max\left(\left|\frac{\partial^2 x}{\partial v^2}\right|, \left|\frac{\partial^2 y}{\partial v^2}\right|, \left|\frac{\partial^2 z}{\partial v^2}\right|\right)\right]. \tag{15}$$

To compute the bounding boxes for a NURBS surface, we first evaluate the surface $S(u,v)$ in a grid of points using our NURBS evaluator on the GPU. We also evaluate the precise first derivatives of the surface, $\partial S/\partial u$ and $\partial S/\partial v$, at these points as explained in Section 3. We approximate the second partial derivatives of the surface by central differencing (explained below in Section 4.1). We then find the value of K for the surface using (12). The bounding boxes themselves are constructed by constructing boxes that enclose sets of four adjacent surface points and then expanding this box by K, which ensures that no part of the surface penetrates out of the bounding box.

4.1 Curvature Evaluation

Evaluating the exact curvature of the surfaces along the two parameter directions can be performed in a similar manner to evaluating the first derivatives. However, the number of additional calculation steps (16 passes for a bicubic surface) required for this operation is prohibitively many and therefore cannot be completed in a real-time setting. Nevertheless, since we have exact derivatives along the two parameter directions, we can approximate the second derivatives to a

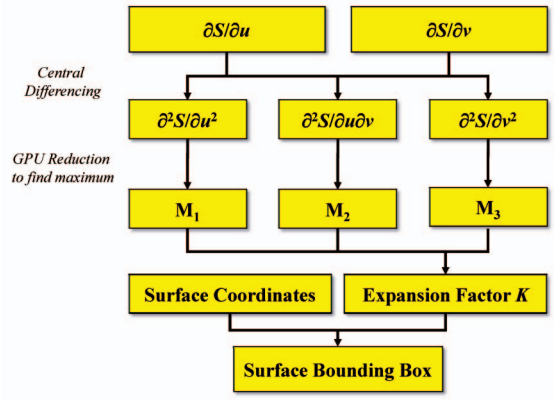


Fig. 5. Flow of algorithm to evaluate bounding boxes of a NURBS surface on the GPU.

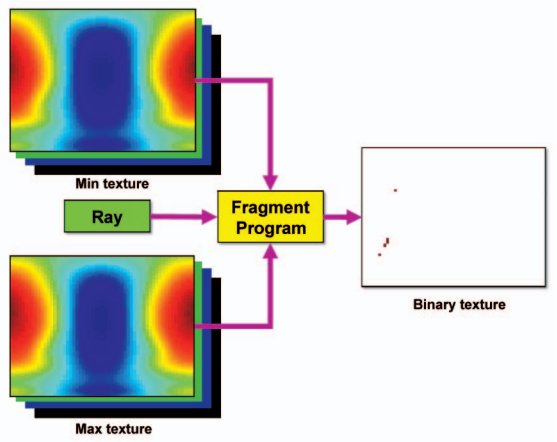


Fig. 6. Bounding-boxes stored as min and max textures are tested with the ray using a fragment program; its output is a binary texture indicating the intersection.

reasonable accuracy (error $< O(1/n^2)$ for n evaluation points) by evaluating them using central differencing.

The central differencing formula for evaluating the second derivatives is given in (16). The value of h is $1/n$ for the u direction and $1/m$ in the v direction since the surface is evaluated on a $(n+1) \times (m+1)$ grid of evaluation points. Three second-derivative values have to be calculated for each surface point: the second derivatives with respect to each parameter direction ($\partial^2 S/\partial u^2$ and $\partial^2 S/\partial v^2$) and one mixed second derivative ($\partial^2 S/\partial u \partial v$). However, we can use our same fragment program written to perform the central differencing operation to evaluate the second derivatives with different first derivative textures as input. For example, (17) shows how to calculate the second derivative with respect to u using the first derivative as input using central differencing:

$$\frac{\partial F(x)}{\partial x} = \frac{F(x+h) - F(x-h)}{2h}, \tag{16}$$

$$\frac{\partial^2 S}{\partial u^2} = \frac{\partial\left(\frac{\partial S(u,v)}{\partial u}\right)}{\partial u} = \frac{\frac{\partial S(u+h,v)}{\partial u} - \frac{\partial S(u-h,v)}{\partial u}}{2h}, \quad h = \frac{1}{n}. \tag{17}$$

To perform this operation on the GPU, we first evaluate the constants M_1, M_2, and M_3 on the GPU (Fig. 5). We calculate these constants by first evaluating the three second-derivatives as explained above for each point on the grid in parallel using a fragment program written to evaluate the second derivatives. We then find the maximum value of each derivative; unfortunately, such a "reduction" operation cannot be performed in a single pass on current GPUs but requires $\log n$ passes for an $n \times n$ texture. We use the constants, M_1, M_2, and M_3, to find the expansion factor K for the surface, which is constant for a given surface patch. Finally, we construct the bounding boxes by first using sets of four adjacent surface points to get an AABB and then expand this box in all three directions by K. The bounding boxes themselves are stored as two textures, one each for the two extreme corners of the AABB, on the GPU. We evaluate all the bounding boxes for a surface in a single pass using a fragment program written to evaluate the bounding boxes, which then outputs the values to the two different textures using multiple render targets.

5 Inverse Evaluation of NURBS Surfaces

Given a point that lies on the NURBS surface, inverse evaluation is the process of finding the parameter values corresponding to that point. Since the B-spline basis functions are nonlinear, theoretical expressions for the inverse evaluation are very complex and differ based on the degree of the surfaces. Therefore, inverse evaluations are usually performed numerically to find a solution within a desired tolerance.

The standard numerical approaches based on solving the NURBS equations for inverse evaluation are not easily parallelizable to be performed efficiently on the GPU. Therefore, we chose a method based on axis-aligned bounding boxes. The AABBs for the NURBS surface are constructed using the method outlined in Section 4. In the case of selection and directly drawing on the surface, the AABBs are aligned parallel to the ray cast in the viewing direction, through the current location of the mouse. We then check for intersection between the ray and all the AABBs simultaneously using a fragment program written to perform this intersection test. The output of this program is a two-dimensional array of binary values with the value 1, indicating the intersection of the ray with the corresponding AABB (Fig. 6). In addition, the intersecting AABB also contains information about the minimum and maximum parameter values of the surface subpatch enclosed by the AABB. Using this correspondence, we can efficiently find the parametric (u, v) value corresponding to the ray intersection point on the surface.

Since the NURBS surfaces are usually curved, there can be many surface subpatches intersecting the given ray. We find the addresses of all the intersecting bounding boxes (locations with the value 1 in the binary texture) by using the GPU stream reduction operation explained in Section 5.1. We use this address to access information about the intersecting bounding box as well as the parametric ranges of the surface subpatch enclosed by the bounding box. Using the bounding box information, we get bounds on the location of the intersection point of the ray with the surface in both the model space as well as in the parametric space simultaneously. If the bounding boxes are smaller

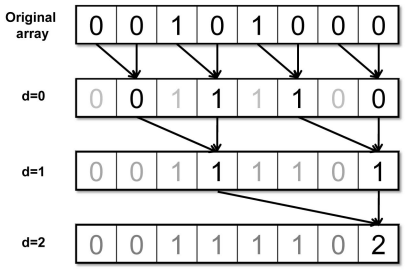


Fig. 7. Example of the up-sweep operation performed on a 1D array given in the first row. The inputs indicated are summed at each step.

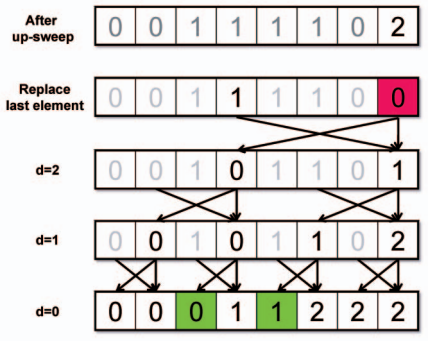


Fig. 8. Example of the down-sweep operation performed on the original 1D array given in Fig. 7. The elements corresponding to the values of 1 in the original input are highlighted in the result; these are the addresses where those values are to be scattered.

than the required tolerance, we can take the midpoint of the bounding box as the intersection point of the ray with the surface. Once all the ray intersection points on the surface are found, we output only the point that is closest to the view plane by evaluating the distance of all the ray intersection points from the view plane on the CPU and choosing the point with the smallest distance value.

5.1 GPU Stream Reduction

An essential operation in our inverse evaluation algorithm is to find the addresses of all the bounding boxes that intersect with a particular ray so that we can use this address to access information about the intersecting bounding box. In this operation, we find the indexes (location) of the texels in the texture that have the given value (in this case, 1). This corresponds to a class of problems known as non-uniform stream reduction. Stream reduction is usually considered a serial operation since the number of elements in the output is not known, and hence, the whole input has to be operated upon to output the correct result. We build on previous work that developed parallel algorithms based on parallel prefix sum for this operation, which we summarized in Section 2. Implementing this parallel prefix sum on standard graphics hardware is not straightforward, however, due to the lack of scatter functionality on standard programmable GPUs.

We first explain briefly the parallel stream reduction operation described in [16]. It consists of three main steps: up-sweep, down-sweep, and scatter. The up-sweep operation computes a hierarchy of $\log n$ levels, where each element at a higher level is obtained as a sum of two elements in the lower-level (Algorithm 1). An example of the up-sweep operation is shown using an eight-element 1D array (Fig. 7). The last element at the end of the operation gives the total number of elements with the value 1 in the input array. After performing this operation, we obtain a binary tree with the last element as the root node and the original array as the leaf nodes; each node of this tree represents the sum of all the values in the subtree of that node.

for $d = 0$ *to* $\log_2 n - 1$ **do**
  forall $k = 0$ *to* $n - 1$ *by* 2^{d+1} **in parallel do**
    $x[k+2^{d+1}-1] \longleftarrow x[k+2^{d}-1]+x[k+2^{d+1}-1]$
  end
end

Algorithm 1: The up-sweep algorithm to construct a hierarchy of the input.

The down-sweep operation given by Algorithm 2, performed on the array resulting from Algorithm 1, computes the exclusive prefix sum of the original input array. The exclusive prefix sum of an array is defined as the sum of all the values preceding a particular position in the array not including the value in the position itself. Fig. 8 gives an example of the down-sweep operation performed on the output shown in Fig. 7 in order to calculate the exclusive prefix sum for the original input given in Fig. 7. The first step of the down-sweep operation is to replace the last element (root element) in the array obtained after the up-sweep operation with the value 0. Then in the consecutive steps, the parent element at each subarray is copied to the left element of the child array and the right element of the child array is calculated as the sum of the old left element and the parent element. In effect, every element now contains the sum of all the elements to the left of itself in the tree structure.

The value of the exclusive prefix sum at the positions where the value of the input array is 1, gives the address to which that particular input value has to be scattered to perform the stream reduction. The final step, after the up-sweep and down-sweep are completed, is the scatter operation in which this address is used to reduce the input stream such that the elements with value 1 are collected at the front of the array.

$x[n-1] \longleftarrow 0$
for $d = \log_2 n - 1$ *down to* 0 **do**
  forall $k = 0$ *to* $n - 1$ *by* 2^{d+1} **in parallel do**
    $t \longleftarrow x[k+2^{d}-1]$
    $x[k+2^{d}-1] \longleftarrow x[k+2^{d+1}-1]$
    $x[k+2^{d+1}-1] \longleftarrow t + x[k+2^{d+1}-1]$
  end
end

Algorithm 2: The down-sweep algorithm to construct the inclusive prefix sum.

However, we cannot directly use this stream reduction algorithm on the GPU due to three issues. The first issue is that the original algorithm was developed for one-dimensional arrays, and hence, has to be adapted to operate on a two-dimensional texture. The second issue is that the traditional GPGPU model based on OpenGL or DirectX does not allow the scatter operation, which is the last step of the stream reduction algorithm. Finally, the original

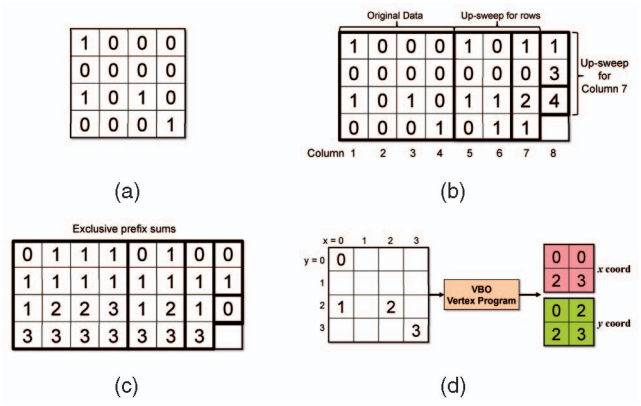


Fig. 9. Different steps of the GPU stream reduction algorithm. (a) Input, (b) Up sweep, (c) Down sweep, and (d) Scatter using VBO.

formulation in [16] computed the prefix sum in situ by modifying the input array. This is not possible using the standard GPGPU framework since we cannot read and write to the same location simultaneously.

We solve the first problem by first assuming that each row of the texture is a separate array and compute the first part of the up-sweep operation until each row array is reduced to a single element. Now we again perform the up-sweep operation on the array formed by concatenating all the single elements in a column along the column direction. In the example shown in Fig. 9b, we perform the up-sweep operation on each row until we end up with the values in column 7. Then we perform the up-sweep operation on column 7 and output the results to column 8. As shown in the example, to overcome the restriction of reading and writing to the same memory location, we maintain a hierarchy of the input texture. This method uses only twice the storage as the original texture used, and a single fragment program written to perform the summation can be repeatedly used. We compute the up-sweep operation in $O(\log n)$ passes.

We then perform the down-sweep operation in a similar manner but in reverse order, by first performing the operation along the columns and then extending it to the rows to obtain the exclusive prefix sum of the input. In the example shown in Fig. 9c, each bold box contains the exclusive prefix sum of the corresponding bold box in Fig. 9b.

Once we have the output from the down-sweep operation, we extract the address of only those texels that have the value 1 in the input texture (Fig. 9d). We reinterpret this texture as a VBO and use a vertex program, written to output the addresses of the input values with value 1 as (x, y) coordinates, to write to two separate channels of the output texture. The size of the output texture varies based on the number of elements with value 1 in the input texture; it is equal to the first square number larger than the number of elements with value 1 in the input. This output texture is then directly used by the inverse evaluation and the surface-surface intersection applications for further processing.

5.2 GPU Implementation of Inverse Evaluation

The algorithm used for performing the full inverse evaluation is given pictorially in Fig. 11. The three steps in the top row of Fig. 11—evaluating the surface, constructing bounding boxes, and finding intersecting boxes—are performed on the GPU. The data corresponding to the selected bounding box are read back from the GPU. We then check on the CPU whether the ranges in the parametric domain of the surface as well as the size of the bounding box are within the required tolerance; for example, we can use an absolute tolerance of 10^{-6} in the parametric space and a relative tolerance of 10^{-3} in the model space. If the tolerance conditions are met, we output the midpoint of the parametric range as the output of the inverse evaluation. If not, we reevaluate the NURBS surface at a finer resolution within the previously output parametric range(s). These tolerances are usually met within two or three iterations since we evaluate the surface at a high resolution $(1,024 \times 1,024)$ during each iteration.

5.3 Applications of Inverse Evaluation

We can build different modeling operations using the inverse evaluation algorithm as the basic module. These operations include ray intersections, direct sketching on NURBS surfaces, and interactive trimming. Fig. 10a shows an example where we compute all the intersection points (two in this case, marked with red crosses) of a particular ray with the surfaces of a toy model. By aligning the ray direction perpendicular to the view plane, we can use the same algorithm for selecting a particular surface from a given set of NURBS surfaces.

One of the most important advantages of a real-time algorithm to perform inverse evaluation is the ability to sketch directly on the NURBS surface. The advantage comes from the fact that the curve is simultaneously sketched both in the three-dimensional model space as well as in the two-dimensional parameter space. This helps in performing

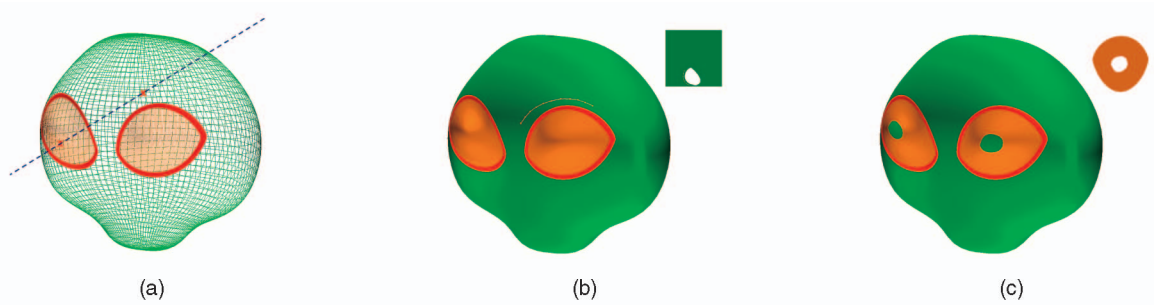


Fig. 10. Different NURBS modeling applications using inverse evaluation. (a) Ray intersection, (b) sketching directly on the surface, and (c) interactive trimming: the eyes of the model were trimmed interactively.

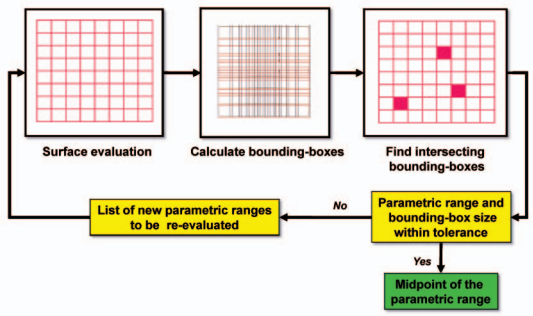


Fig. 11. Algorithm for inverse evaluation of NURBS surfaces.

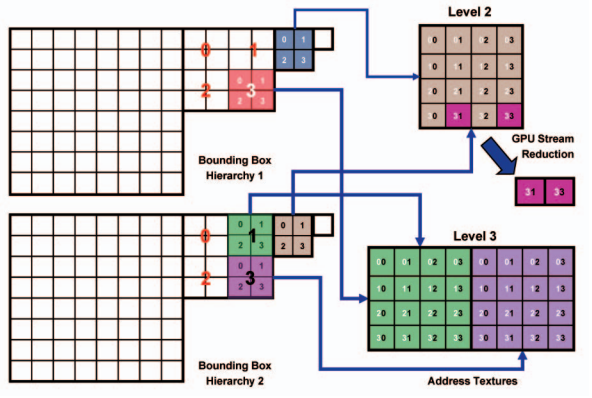


Fig. 12. Example of hierarchical bounding box comparison in the surface-surface intersection algorithm.

modeling operations like extrusions and trimming, where the parameter space sketches are typically used for defining these operations. Fig. 10b shows a curve sketched on a NURBS model and the curve in the parametric domain is shown in the inset.

By combining our sketching interface with the algorithm that renders trimmed NURBS surfaces in real time, we can perform interactive trimming operations (Fig. 10c). Using our interactive trimming application, the designer gets immediate feedback on the result of the trimming operation, unlike current commercial CAD systems.

6 NURBS Intersection Curve Evaluation

Calculating the intersection curve of a surface-surface intersection is a frequently encountered operation in CAD systems. It forms an essential part of important CAD operations like trimming, filleting, and b-rep generation from Boolean operations. However, since it is a slow operation, it is usually performed in the background, and thus, the user does not get real-time feedback except in the simplest of cases. We present a GPU-accelerated surface-surface intersection algorithm to calculate intersection curves both in the model space as well as in the parametric spaces of both the surfaces.

We now give a broad overview of our surface-surface intersection algorithm. Our algorithm makes use of bounding box hierarchies to accelerate the intersection operation. We evaluate both intersecting surfaces using the GPU and then use the method described in Section 4 to construct the AABBs for the surfaces, using the same coordinate frame. We construct a hierarchy of bounding boxes by combining four bounding boxes at one level to construct a single bounding box in the next level. To find the intersection curve, we then traverse along the hierarchy simultaneously for both the surfaces and find the intersecting bounding boxes in the lowest level using the GPU. At the same time, we also get the ranges in the parametric domain corresponding to the intersecting surface patches. We then check if the sizes of the bounding boxes as well as the parametric ranges are within a user-defined tolerance. Once the tolerance conditions are met, we get a better estimate of the point on the intersection curve by intersecting the linearized surface patch within the intersecting bounding boxes.

We will explain the details of our surface-surface intersection algorithm with an example (Fig. 12). Given two surfaces, S_1 and S_2, we evaluate them and construct their bounding boxes as explained in previous sections. We also construct the bounding box hierarchies for both the surfaces and store them on the GPU as textures. Once we have the hierarchies, we use the CPU to test whether the bounding boxes of the surfaces intersect at the topmost level, level 1. If so, we then test the bounding boxes from the next level onward on the GPU, using one pass per level. We perform the intersection tests for all the bounding boxes in a level in parallel using a fragment program written to perform the bounding box intersection test. The input to the fragment program is a texture called the address texture that contains the address of the bounding boxes in the hierarchy (also stored as textures). For example, to test for intersection in the second level, we make use of a 4×4 address texture on the GPU, where we test for intersection of a bounding box of S_1 with all the four bounding boxes of S_2. In Fig. 12, the rows of the address texture (Level 2) correspond to bounding boxes from S_1 and the columns correspond to bounding boxes from S_2. The address texture is a four-component texture consisting of the address corresponding to bounding boxes of S_1 and S_2 in the bounding box hierarchy textures ((u_1, v_1, u_2, v_2) stored using RGBA channels). The intersection test is performed on the GPU using a fragment program, which uses the address information to retrieve the data for the bounding boxes from the bounding box hierarchy and subsequently tests them for intersection. The output of the fragment program is a binary texture with a value of 1 indicating an intersection. We use the stream reduction algorithm explained in Section 5.1 to find the address of the intersecting bounding boxes. In the example shown, we find that bounding box 3 of S_1 intersects with bounding boxes 1 and 3 of S_2 at level 2.

In the next level (pass), we test for the intersection of the children of the intersecting bounding box pairs of the previous level simultaneously on the GPU. Thus, the size of the address texture varies dynamically based on the number of intersections in the previous levels. The size of the address texture is always a multiple of 4 since we test for intersection between S_1 and S_2 in blocks of 4×4 intersection tests. However, we make sure that this is a square texture and its size is a power-of-2 to optimize the stream reduction algorithm. The parallelism of the GPU is exploited in checking for intersection of all the intersecting

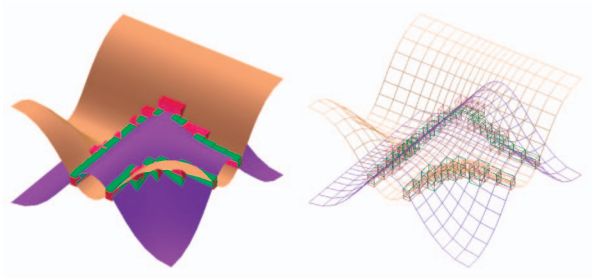


Fig. 13. Intersecting bounding boxes of two NURBS surfaces.

bounding box pairs at any given level and this helps in accelerating the intersection algorithm as the address texture grows in size. Once we reach the base level of the bounding box hierarchy, we get a list of the bounding boxes that intersect at this level (Fig. 13). This list can then be used for further processing on the CPU to get the actual intersection curve.

In addition, we use this list to render the points on the intersection curve of each surface to a dynamic texture in the parametric domain. We map this texture back onto each surface, providing real-time feedback to the designer about the shape of the intersection curve (Fig. 14).

6.1 Fitting an Intersection Curve

To get a better estimate of the intersection point lying on the intersection curve of two surfaces, we intersect the surface subpatches enclosed by the intersecting bounding boxes on the CPU. We approximate each surface subpatch inside the bounding box with two triangles that share an edge. We intersect these two triangles contained inside the bounding box of the first surface with the two other triangles contained in the bounding box of the second surface. This gives rise to four pairs of intersection tests between the triangles of the two surfaces; each intersection test can be true or false, generating 16 different cases. We show one particular case in Fig. 15, where one triangle of surface $S1$ intersects with another triangle of surface $S2$. The four triangles are denoted as $A_0A_1A_2$ and $A_1A_2A_3$ for surface S_1, and $B_0B_1B_2$ and $B_1B_2B_3$ for surface S_2 in the figure. We find the midpoint of the intersection line-segment and use this midpoint as a point on the intersection curve if it lies within the intersecting region of the bounding boxes. The intersecting region of the bounding boxes is denoted by $(x_{min}, y_{min}, z_{min})$ and $(x_{max}, y_{max}, z_{max})$ in the figure. In the case of multiple intersections, we take the centroid of the

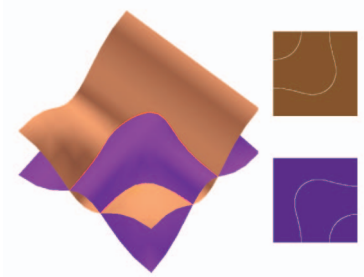


Fig. 14. Intersection curves of two NURBS surfaces plotted both in the model space as well as in their corresponding parametric spaces.

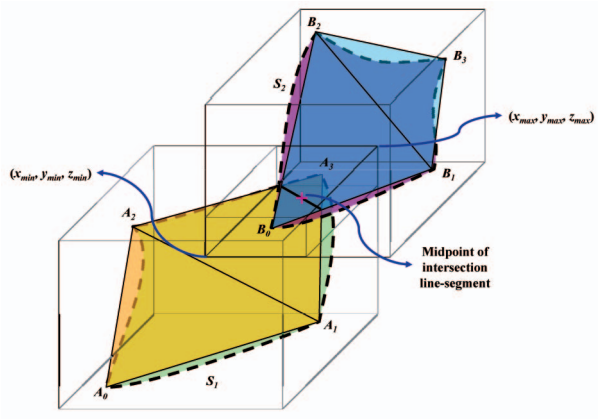


Fig. 15. Intersecting triangles inside overlapping bounding box pairs to get a better estimate of the point on the intersection curve.

midpoints of the intersection line-segments computed for each intersecting triangle pair as a point on the intersection curve. Only if this centroid lies inside the intersecting region of the bounding boxes do we use this point for fitting a curve.

We then extract the seven-tuple $(x, y, z, u_1, v_1, u_2, v_2)$ for each point found on the intersection curve using the above method, where (x, y, z) is the point on the intersection curve in 3D space, (u_1, v_1) and (u_2, v_2) are the corresponding points in the parametric space of surfaces S_1 and S_2, respectively. The parametric points are found by computing the barycentric coordinate of the (x, y, z) intersection point in each of the corresponding intersecting triangles and then interpolating the parametric coordinates at the three vertices of the triangle linearly using the barycentric coordinates.

Input : List of points on the intersection curves in $\Re^7$.
Output: Polyline list L, corresponding to the intersection curves (an ordered list of connected edges).

1. Make all points into a polyline of length 0; add to L.
2. For all polylines in L, find the pair, P_1 and P_2, that is the closest (between two end points of P_1/P_2 in $\Re^7$).
3. If distance greater than maximal distance to merge Quit;
Otherwise,
(a) Merge P_1 and P_2 into a new polyline P.
(b) Replace P_1 and P_2 by P in L.
(c) Goto 2.

Algorithm 3: Algorithm to fit polylines to the points on the intersection curves.

Finally, to compute the actual intersection curves themselves from the list of points, we compared two different algorithms. The first one is a greedy algorithm (Algorithm 3) that computes the intersection curves by successively merging polylines that are close to each other. We work in the seven-dimensional space $\Re^7$, integrating the data from both the model space as well as the two parametric spaces. Performing the curve fitting in $\Re^7$ is more robust since different components of intersection curves that might be close in a particular geometric or parametric space are less likely to be simultaneously close in all three spaces.

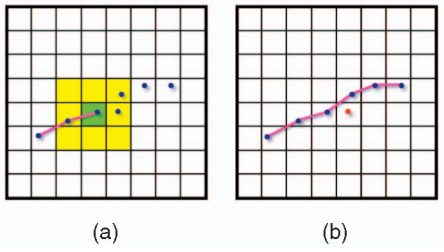


Fig. 16. Example showing the possible generation of single-point polyline by Algorithm 4. (a) The one-ring of bounding boxes (shaded). (b) The point marked in red is not merged and is output as a single-point polyline if it is not the closest point in $\Re^7$.

The second algorithm (Algorithm 4) uses the fact that the intersection points we find are enclosed by AABBs that are part of a regular grid. We can thus fit a polyline by connecting a point to the closest point whose enclosing bounding box is a neighbor to the enclosing bounding box of the current point, limiting our search to the one-ring neighborhood of bounding boxes. We still find the closest point in $\Re^7$. After adding the closest point in the one-ring neighborhood to the polyline, we repeat our search to find another point that is the closest to the point just added. Since the starting point can be in the middle of an intersection curve, we have to grow the polyline in both directions. This algorithm can be compared to a depth-first search on a list to find all the connected components and hence takes $O(n)$ time. However, if there is more than one remaining adjacent bounding box with unmerged intersection points, some points may not be merged into the polyline and will be output as polylines of length 1 (Fig. 16). These polylines can then either be discarded or merged at the correct position of the longer polylines by making an additional pass.

Input : List of points on the intersection curves in $\Re^7$.
Output: Polyline list L, corresponding to the intersection curves (an ordered list of connected edges).

1. Add all the points to unmerged points list M.
2. Add S, the first point in M to a new polyline P.
3. $A \longleftarrow S$
4. While there are unmerged points in the 1-ring of A (the point added last to P)
  (a) Find the closest point B in the 1-ring of A.
  (b) Add B to tail of P and remove it from M.
  (c) $A \longleftarrow B$
5. $A \longleftarrow S$
6. While there are unmerged points in the 1-ring of A
  (a) Find the closest point B in the 1-ring of A.
  (b) Add B to head of P and remove it from M.
  (c) $A \longleftarrow B$
7. If M is empty
    Quit;
  Otherwise,
  (a) Add P to L.
  (b) Goto 2.

Algorithm 4: Faster algorithm to fit polylines to the points on the intersection curves.

The time taken to fit a polyline using Algorithm 3 depends on an efficient closest neighbor query. Currently, we perform this operation through an exhaustive search that takes $O(n^2)$ time, which could be optimized by using more efficient search techniques, but we would still expect it to be slower than the $O(n)$ time Algorithm 4. For the example shown in Fig. 14, the polyline fitting for over 7,000 points takes 0.20 seconds on a 2-GHz PC for a tolerance value of 2×10^{-3}. On the other hand, the time taken by a single pass of Algorithm 4 was 0.02 seconds for the same input and tolerance value. However, 320 single-point polylines were also produced by Algorithm 4, which were discarded. From a tolerant geometry point of view, discarding these points does not reduce the overall tolerance achieved compared to Algorithm 3.

Since our input list of points on the intersection curve is sufficiently dense, a polyline that passes through these points can be directly used for further modeling operations. If a more compact representation is required, we can fit a NURBS curve of any required order that approximates the points on the intersection curve using standard curve-fitting techniques. Since the intersection points obtained from our algorithm are enclosed within their corresponding bounding boxes both in the model space and in the parametric space, we can guarantee a required bound on the results. In addition, if the arbitrary user-defined bounds are small enough, we are guaranteed not to miss any portion of the intersection curve. Since we also give instantaneous visual feedback to the user, the user will immediately know if there are any features missing and can reduce the tolerance to obtain the desired result.

One of the main limitations of both our algorithms for fitting a polyline is that they will fail to recreate the correct topology when two unrelated intersection curves are very close on both surfaces. This can happen when an intersection curve splits into two branches or when the two surfaces are locally flat and are touching each other. A method that ensures the topology of the intersection set is to be sought, possibly at the CPU level, using the GPU only to find the simple intersection curves. Such a method will also help in balancing the load between the CPU and the GPU.

6.2 Self-Intersection Evaluation

We extended our surface-surface intersection algorithm to detect and evaluate self-intersections in NURBS surfaces. To perform the self-intersection test, we create two instances of the bounding box hierarchy for the surface on the GPU. We then test for intersection between these two surface instances using the same GPU algorithm we use to perform surface-surface intersections. The output of this algorithm is a list of bounding box pairs at the lowest level of the hierarchy that overlap each other. We then remove from this list all the pairs which correspond to the same surface subpatch. Finally, if there are any bounding box pairs which belong to different surface subpatches left in the list, then the surface is self-intersecting.

Once we find a surface to be self-intersecting, we perform triangle-triangle intersection of the triangles contained within the intersecting bounding box pairs. Similar to the surface-surface intersection algorithm, we find points on the self-intersection curve and then fit a polyline through this self intersection curve. However, the main limitation of the algorithm is that a self intersection smaller than the tolerance will be rejected. This can occur in a local self-intersection due

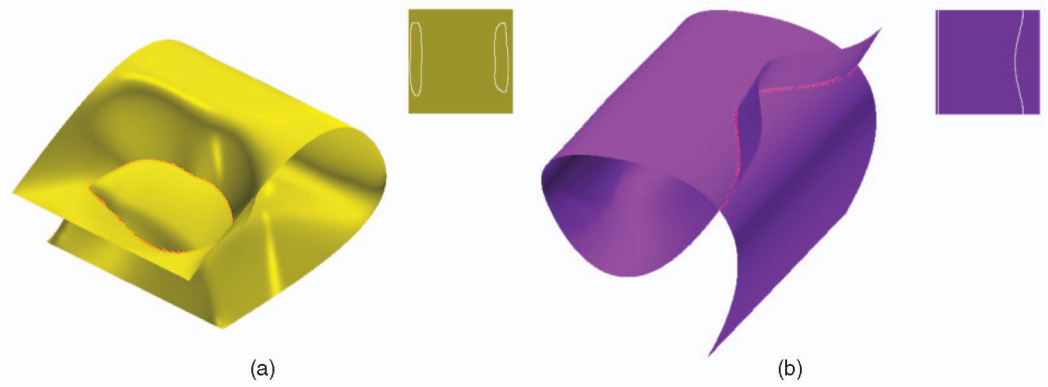


Fig. 17. Detection and evaluation of self-intersections in NURBS surfaces.

to curvature in an offset surface, and a complete intersection loop will be difficult to evaluate since the tolerance needs to be infinitesimally small in this case. Fig. 17 shows two examples where we detect and evaluate self-intersection curves in NURBS surfaces. The example shown in Fig. 17b took 0.42 seconds to compute the self-intersection curves to a tolerance value of 2×10^{-3}, while the more complicated example shown in Fig. 17a took 0.97 seconds.

6.3 Intersection Timing

We timed our GPU-accelerated algorithm for evaluating the intersection curves on a 3-GHz CPU with 2 GB of RAM equipped with a NVIDIA Quadro FX4500 GPU with 512 MB graphics memory running Windows XP. We performed a surface-surface intersection of the two NURBS surfaces shown in Fig. 18. The surfaces were bicubic NURBS with 403×199 and 298×313 control points, respectively. We used Algorithm 4 to fit the polylines during the timing. We compare our timings to evaluate the intersection curves to the required user-defined tolerance with those of the commercial solid modeling kernel ACIS (v18).

Fig. 19 compares the time for evaluating the intersection curves by varying the tolerance values. Our GPU-accelerated evaluation is more than 40 times faster than ACIS in computing the intersection curves to the standard tolerance of 10^{-3} used in ACIS. The output from ACIS is an interpolated polyline where the points on the polyline are within the user-defined tolerance value from the exact intersection curve. ACIS does not guarantee any tolerance on the piecewise linear line segments that make up the polyline [32]. On the other hand, we evaluate dense intersection points with their spacing adjusted based on the tolerance to achieve a guaranteed tolerance on the piecewise linear segments of the polyline as well. We compute almost 40 times as many points on the intersection curve as ACIS does for the standard ACIS tolerance value of 10^{-3} (Fig. 20).

Table 1 gives the breakdown of the timing of our intersection algorithm for evaluating the intersection curves, shown in Fig. 14, for a tolerance value of 10^{-3}. The evaluation of the NURBS surfaces is a large fraction of the total time. Note that we do not require such high-tolerance values for giving visual feedback; hence, it can be performed at interactive rates.

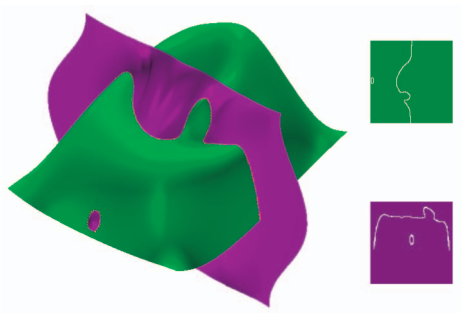


Fig. 18. NURBS surfaces used for timing the evaluation of intersection curves.

7 Conclusions

We present fast algorithms to perform interactive modeling operations on NURBS surfaces. Our algorithms do not require the latest graphics cards and are backward compatible with any graphics card that has basic programming capabilities. This is essential for the actual adoption of our algorithms in commercial CAD systems. We expect the performance of our algorithms to only improve with the advent of new and faster graphics cards.

Both our GPU algorithm to sketch on NURBS surfaces as well as our GPU-accelerated algorithm to calculate intersection curves give real-time feedback to the designer about the shape of the curves in the parametric space. This gives a direct handle for the designer to check for inconsistency if

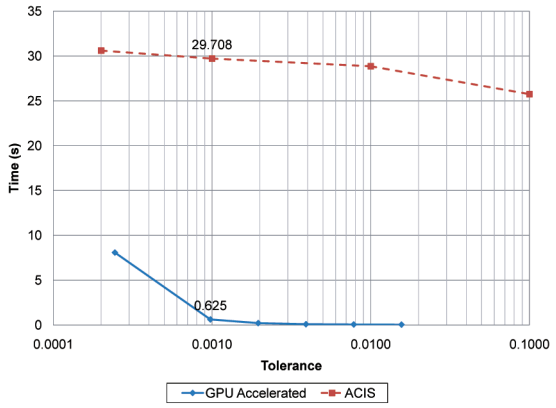


Fig. 19. Time taken for evaluating the intersection curves of the two NURBS surfaces shown in Fig. 18 with different resolutions. Note that we are evaluating many more points on the intersection curve for a given resolution (Fig. 20).

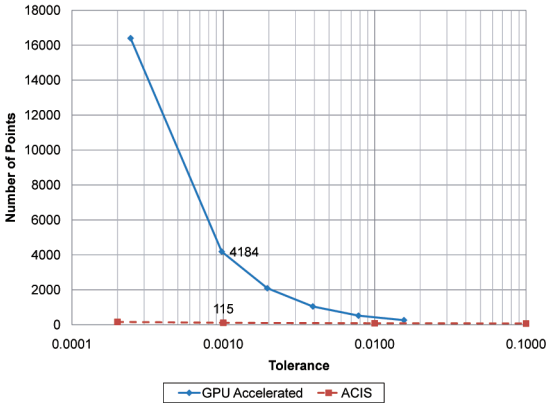


Fig. 20. Number of points evaluated on the intersection curve for different resolutions.

TABLE 1
Breakdown of the Timing

Operation	Time(s)
Evaluate NURBS surfaces	0.27
Perform intersection tests	0.05
Calculate dense intersection points	0.02
Fitting the polyline	0.14
Total	0.48

Breakdown of the timing to perform different operations of our intersection algorithm. The values are for the example shown in Fig. 14 for a tolerance value of 10^{-3}.

models fail during rebuilds in a CAD system. Our interactive trimming tool helps the designer to easily interact with and edit the NURBS models. Moreover, the applications that we have outlined in our paper form only a small part of the different kinds of applications that can be developed with the help of GPU-accelerated basic modeling operations. There is large potential for developing diverse applications that use these operations as building blocks.

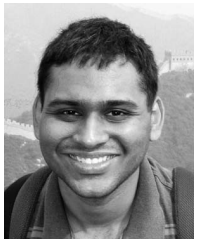
ACKNOWLEDGMENTS

The authors would like to thank Wei Li for helping them in the background research for parallel stream reduction and Professor Stephen Mann for helping them in NURBS derivative computations. They would also like to thank NVIDIA and AMD for providing them with their hardware. This material is based upon work supported in part by SolidWorks Corporation, UC Discovery under Grant No. DIG05-10190, and the US National Science Foundation under Grant No. 0547675.

REFERENCES

[1] A. Krishnamurthy, R. Khardekar, and S. McMains, "Direct Evaluation of NURBS Curves and Surfaces on the GPU," *Proc. ACM Symp. Solid and Physical Modeling,* pp. 329-334, 2007.
[2] T. Kanai, "Fragment-Based Evaluation of Non-Uniform B-Spline Surfaces on GPUs," *Computer-Aided Design and Applications,* vol. 4, no. 3, pp. 287-294, 2007.
[3] M. Guthe, A. Balazs, and R. Klein, "GPU-Based Trimming and Tessellation of NURBS and T-Spline Surfaces," *ACM Trans. Graphics,* vol. 24, no. 3, pp. 1016-1023, 2005.
[4] H. Pabst, J. Springer, A. Schollmeyer, R. Lenhardt, C. Lessig, and B. Froehlich, "Ray Casting of Trimmed NURBS Surfaces on the GPU," *Proc. IEEE Symp. Interactive Ray Tracing,* pp. 151-160, 2006.
[5] C. de Boor, *A Practical Guide to Splines.* Springer-Verlag, 1978.
[6] C. Loop and J. Blinn, "Resolution Independent Curve Rendering Using Programmable Graphics Hardware," *Proc. ACM SIGGRAPH '05,* pp. 1000-1009, 2005.
[7] M. Guthe, A. Balazs, and R. Klein, "GPU-Based Appearance Preserving Trimmed NURBS Rendering," *J. WSCG,* vol. 14, 2006.
[8] D.L. Toth, "On Ray Tracing Parametric Surfaces," *Proc. ACM SIGGRAPH '85,* pp. 171-179, 1985.
[9] T. Nishita, T.W. Sederberg, and M. Kakimoto, "Ray Tracing Trimmed Rational Surface Patches," *Proc. ACM SIGGRAPH '90,* pp. 337-345, 1990.
[10] W. Martin, E. Cohen, R. Fish, and P. Shirley, "Practical Ray Tracing of Trimmed NURBS Surfaces," *J. Graphics Tools,* vol. 5, no. 1, pp. 27-52, 2000.
[11] T.J. Purcell, I. Buck, W.R. Mark, and P. Hanrahan, "Ray Tracing on Programmable Graphics Hardware," *ACM Trans. Graphics,* vol. 21, no. 3, pp. 703-712, 2002.
[12] T.J. Purcell, C. Donner, M. Cammarano, H.W. Jensen, and P. Hanrahan, "Photon Mapping on Programmable Graphics Hardware," *Proc. ACM SIGGRAPH/EUROGRAPHICS Conf. Graphics Hardware (GH '03),* pp. 41-50, 2003.
[13] N.A. Carr, J. Hoberock, K. Crane, and J.C. Hart, "Fast GPU Ray Tracing of Dynamic Meshes Using Geometry Images," *Proc. Graphics Interface '06,* pp. 203-209, 2006.
[14] G. Elber and M.-S. Kim, "Geometric Constraint Solver Using Multivariate Rational Spline Functions," *Proc. Sixth ACM Symp. Solid Modeling & Applications,* pp. 1-10, 2001.
[15] T. Thompson and E. Cohen, "Direct Haptic Rendering of Complex Trimmed NURBS Models," *Proc. Eighth Ann. Symp. Haptic Interfaces for Virtual Environment and Teleoperator Systems,* 1999.
[16] G.E. Blelloch ed., *Vector Models for Data-Parallel Computing.* MIT Press, 1990.
[17] D. Horn, *GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation,* chapter on Stream Reduction Operations for GPGPU Applications. Addison-Wesley, 2005.
[18] A. Greß, M. Guthe, and R. Klein, "GPU-Based Collision Detection for Deformable Parameterized Surfaces," *Computer Graphics Forum,* vol. 25, no. 3, pp. 497-506, 2006.
[19] S. Sengupta, M. Harris, Y. Zhang, and J.D. Owens, "Scan Primitives for GPU Computing," *Proc. ACM SIGGRAPH/EUROGRAPHICS Symp. Graphics Hardware (GH '07),* pp. 97-106, 2007.
[20] N.K. Govindaraju, S. Redon, M.C. Lin, and D. Manocha, "CULLIDE: Interactive Collision Detection between Complex Models in Large Environments Using Graphics Hardware," *Proc. ACM SIGGRAPH/EUROGRAPHICS Conf. Graphics Hardware (GH '03),* pp. 25-32, 2003.
[21] P. Kipfer, M. Segal, and R. Westermann, "UberFlow: A GPU-Based Particle Engine," *Proc. ACM SIGGRAPH/EUROGRAPHICS Conf. Graphics Hardware (GH '04),* pp. 115-122, 2004.
[22] A. Kolb, L. Latta, and C. Rezk-Salama, "Hardware-Based Simulation and Collision Detection for Large Particle Systems," *Proc. ACM SIGGRAPH/EUROGRAPHICS Conf. Graphics Hardware (GH '04),* pp. 123-131, 2004.
[23] A.A.G. Requicha and J.R. Rossignac, "Solid Modeling and Beyond," *IEEE Computer Graphics Applications,* vol. 12, no. 5, pp. 31-44, 1992.
[24] C.M. Hoffmann, *Geometric and Solid Modeling.* Morgan Kaufmann Publishers Inc., 1989.
[25] N.M. Patrikalakis, "Surface-to-Surface Intersections," *IEEE Computer Graphics and Applications,* vol. 13, no. 1, pp. 89-95, 1993.
[26] 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.
[27] R.E. Barnhill and S.N. Kersey, "A Marching Method for Parametric Surface Surface Intersection," *Computer Aided Geometric Design,* vol. 7, nos. 1-4, pp. 257-280, 1990.
[28] G.A. Kriezis, P.V. Prakash, and N.M. Patrikalakis, "A Method for Intersecting Algebraic Surfaces with Rational Polynomial Patches," *Computer Aided Design,* vol. 22, no. 10, pp. 645-654, 1990.
[29] L.A. Piegl and W. Tiller, *The NURBS Book,* second ed. Springer, 1997.
[30] S.S. Abi-Ezzi and M.J. Wozny, "Factoring a Homogeneous Transformation for a More Efficient Graphics Pipeline," *Computer Graphics Forum,* vol. 9, no. 3, pp. 245-255, 1990.

[31] D. Filip, R. Magedson, and R. Markot, "Surface Algorithms Using Bounds on Derivatives," *Computer Aided Geometric Design*, vol. 3, no. 4, pp. 295-311, 1987.
[32] J. Corney and T. Lim, *3D Modeling with ACIS*. Saxe-Coburg, 2001.

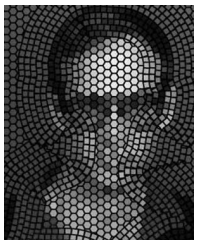


Adarsh Krishnamurthy received the bachelors and masters degrees in mechanical engineering from the Indian Institute of Technology, Madras. He is currently working toward the PhD degree in the Department of Mechanical Engineering at the University of California, Berkeley. His research interests include computer aided Design (CAD), solid modeling, GPU algorithms, computational geometry, and ultrasonic nondestructive testing.

Rahul Khardekar received the PhD degree in mechanical engineering from the University of California, Berkeley. He is currently a 3D graphics engineer in Align Technology, Inc. His research interests include computational geometry, computer graphics, and computer-aided design and manufacturing.

Sara McMains received the PhD degree from U.C. Berkeley in computer science. She is currently an associate professor of mechanical engineering at the University of California, Berkeley. Her research interests include geometric design for manufacturing (DFM) feedback, geometric and solid modeling, CAD/CAM, GPU algorithms, computer-aided process planning, layered manufacturing, and virtual reality.

Kirk Haller received the degree from the University of Wisconsin—Madison. He is currently a director of research at Dassault Systemes SolidWorks Corp., the leader in 3D CAD technology. The SolidWorks' Research Group is focused on advancing science and technology in order to provide better tools for engineers, designers, and other creative professionals. Prior to joining SolidWorks, he was an assistant professor at the University of Waterloo and a research assistant at Dalhousie University.



Gershon Elber received the BSc degree in computer engineering and the MSc degree in computer science from the Technion, Israel, in 1986 and 1987, respectively, and the PhD degree in computer science from the University of Utah, in 1992. He is currently a professor in the Computer Science Department, Technion, Israel. His research interests include computer aided geometric designs and computer graphics. He is a member of the ACM and IEEE. He has served on the editorial board of the Computer Aided Design, Computer Graphics Forum, the Visual Computer, and the International Journal of Computational Geometry & Applications and has served in many conference program committees including Solid Modeling, Shape Modeling, Geometric Modeling and Processing, Pacific Graphics, Computer Graphics International, and SIGGRAPH. He was one of the paper chairs of Solid Modeling 2003 and Solid Modeling 2004, and will be the conference chair of Solid Modeling 2010. He has published over 150 papers in international conferences and journals and is one of the authors of a book titled *Geometric Modeling with Splines—An Introduction*.