International Journal of Shape Modeling
© World Scientific Publishing Company

Real-time Freeform Deformation Using Programmable Hardware

Sagi Schein
HP Labs Israel, Technion, Haifa, 32000 Israel
sagi.schein@hp.com

Gershon Elber
Computer Science Department, Technion, Haifa 32000 Israel
gershon@cs.technion.ac.il

Received (Day Month Year)
Revised (Day Month Year)
Accepted (Day Month Year)

Communicated by (xxxxxxxxxx)

The recent introduction of programmable Graphics Processing Units (GPUs) has had a tremendous impact on real-time graphics. With their extended flexibility, GPUs can support tasks that go way beyond their originally intended rendering functionality. One domain where GPUs can be applicable is in accelerating geometric modeling applications. In this work, we present a real-time, GPU-based, evaluation scheme for trivariate B-spline functions. These functions are used in the implementation of Freeform Deformations (FFDs), a common global deformation algorithm. Using this deformation scheme, complex objects can be freely warped and animated. The proposed scheme was implemented on contemporary commercially-available graphics cards, achieving interactive rates.

Keywords: Trivariate B-spline functions; Freeform Deformations (FFDs); Graphical Processing Units (GPU)

1991 Mathematics Subject Classification : 65D17

1. Introduction and Background

Over the past few years, GPUs have become a powerful computational platform. Designed for vector-oriented computations and employing multiple computational pipelines, GPUs are specifically tailored for real-time rendering. Moreover, current GPUs also offer extensive programmability support that further improves their usability. Consequently, many complex algorithms previously non-implementable in real-time graphics applications, have become viable options. These graphics cards open up a programming interface for two stages in their rendering pipeline, the per-vertex stage, and the per-fragment stage. Nevertheless, even though GPUs are designed for rendering, general purpose computations can also take advantage of

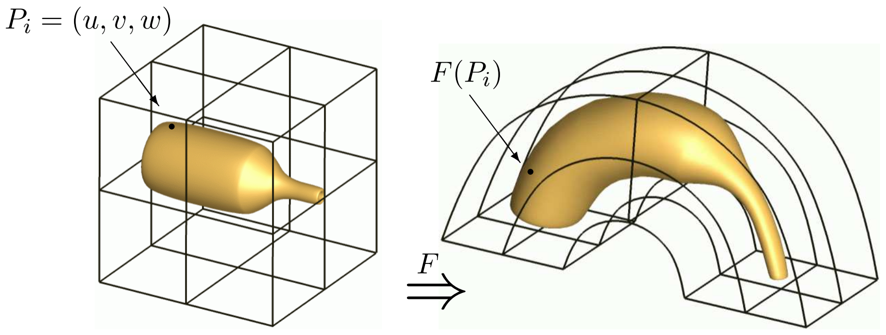


Fig. 1. A trivariate function F is used to map point, P_i, in its parametric domain. The result is a point $F(P_i) \in R^3$.

their high computational throughput. Specifically, expensive geometric modeling operations, which usually precede rendering, could take advantage of the expanded performance.

In this work, we propose a framework for deforming geometric objects on the GPU, at interactive frame rates. There are many available deformation schemes: local, effecting only selected parts of a model, or global. Usually, global methods are more computationally intensive than local ones, due to the larger amounts of geometry that have to be processed. On the other hand, global methods also have several advantages. For example, it is easier to retain the original continuity of a model while deforming it, using global deformation schemes. Among global deformation tools, Freeform Deformations (FFDs) [1] are perhaps the most common. FFDs typically use a trivariate deformation function, $F : D \subset I\!R^3 \Rightarrow I\!R^3$, in the form of trivariate Bézier or B-spline function. These deformation functions define a mapping of an object, $\mathcal{O}$, from D, the parametric domain of the deformation function, F, to the Euclidean space.

The original work on FFDs [1] suggested the use of Bézier functions as the deformation function. In order to improve the flexibility of FFDs, low order trivariate B-spline functions were suggested in [2]. These functions are defined as,

$$F(u, v, w) = \sum_{i,j,k=0}^{l,m,n} P_{ijk} B^o_{i,\boldsymbol{\tau}_\mathbf{u}}(u) B^o_{j,\boldsymbol{\tau}_\mathbf{v}}(v) B^o_{k,\boldsymbol{\tau}_\mathbf{w}}(w), \tag{1}$$

where $P_{ijk} \in I\!R^3$ form the 3D lattice of control points of the trivariate B-spline functions, and $B^o_{i,\boldsymbol{\tau}_u}(u)$ is the i^{th} B-spline basis function of order o over knot sequence $\boldsymbol{\tau}_u$. Throughout this paper, o and $\boldsymbol{\tau}_u$ will be omitted at times to simplify the notation. FFDs can be used to deform geometry in $I\!R^3$, specifically, surface based models. In this context, the deformation is considered to be a composition of a freeform surface $S(u, v) = [x(u, v), y(u, v), z(u, v)]$ and the deformation function, F, such that $\tilde{S} = F(S) = F(x(u, v), y(u, v), z(u, v))$. In this work, we consider only polygonal surface models, where the exact deformation is approximated by deforming the vertices of the models, V_i, as $\tilde{V}_i = F(V_i)$.

The evaluation of trivariate B-spline functions is usually considered a rather time-consuming operation. Each evaluation of a vertex, $V_i \in \mathcal{O}$, requires the evaluation of o^3 B-spline basis functions. As a result, using FFDs inside interactive modeling tools remains limited. To improve the deformation rates of FFDs, hardware acceleration is one option that can be considered. Previous work [3] already outlined some general details of one such solution. In [3], a system for hardware accelerated EFFDs [4], was described. EFFDs is a generalization of FFDs which uses multiple trivariate patches to better approximate the general shape of the deformed model. The system is based on OpenGL's evaluators [5], and suggests a hardware implementation that could support the EFFDs evaluation. This approach is somewhat appealing since it lets the software's API remain the same while leaving the exact hardware subject to future improvements. The only problem with such an approach is that, as of now, most graphics accelerators do not support OpenGL's evaluators in their hardware. Another hardware-based approach for FFDs evaluation [6] implemented it using Field Programmable Gate Array (FPGA). This solution is, however, less suitable for general purpose computer graphics applications, due to its high cost and lack of generality.

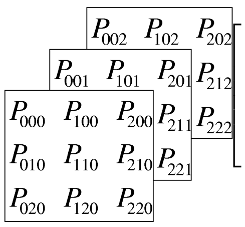
In this work, we present a hardware-based FFDs (HFFDs) evaluation scheme for trivariate B-spline functions. The proposed method is fully capable of real-time deformation of complex models at interactive frame rates using modern GPUs. Further, the proposed scheme can be integrated easily into existing rendering applications, assuming that a proper GPU is available. Finally, the proposed scheme also incorporates all the computational aspects of the evaluation of FFDs inside the GPU.

The rest of this paper is organized as follows. In Section 2, we describe the GPU-based evaluation algorithm for trivariate B-spline functions and consider two evaluation alternatives on the GPU. Section 3 presents some animation and modeling results using HFFDs. Finally, we conclude in Section 4 and consider some future directions.

2. Evaluation of Trivariate B-splines

In order to apply an HFFDs operation, trivariate B-spline functions should be evaluated. As in curves and surfaces, Equation (1) could be represented in a matrix form. Due to the vector oriented architecture and specialized command set of the GPU, vector and matrix operations are much more efficiently processed on the GPU than on the CPU. Hence, the following equation is a rearrangement of Equation (1) so that only vector-matrix multiplications and inner-products are used. In the tri-quadratic case, Equation (1) becomes

$$F(u, v, w) = [B_0(u), B_1(u), B_2(u)]$$



P_{002} P_{102} P_{202}
P_{001} P_{101} P_{201} P_{212}
P_{000} P_{100} P_{200} P_{211} P_{222}
P_{010} P_{110} P_{210} P_{221}
P_{020} P_{120} P_{220}

$$\begin{bmatrix} B_0(v) \\ B_1(v) \\ B_2(v) \end{bmatrix}.$$

$$[B_0(w), B_1(w), B_2(w)] \qquad (2)$$

Let

$$M_k = \begin{bmatrix} P_{00k} \; P_{10k} \; P_{20k} \\ P_{01k} \; P_{11k} \; P_{21k} \\ P_{02k} \; P_{12k} \; P_{22k} \end{bmatrix}, \quad \tilde{M}_k = [B_0(u)B_1(u)B_2(u)]M_k \begin{bmatrix} B_0(v) \\ B_1(v) \\ B_2(v) \end{bmatrix},$$

then,

$$F(u, v, w) = \sum_{k=0}^{2} \tilde{M}_k B_k(w). \qquad (3)$$

The last summation can also be replaced by a single inner product computation. Cubic trivariate B-spline functions can be similarly mapped into four-by-four matrices, which are also directly supported by the GPU.

In Section 2.1, several evaluation schemes for B-spline basis functions in the context of the GPU are discussed. Section 2.2 describes an implementation of an evaluation procedure for quadratic trivariate B-spline functions on the vertex processor, following Equation (3). Unfortunately, this approach is also limited due to some contemporary hardware restrictions, which will be discussed below. In Section 2.3 we suggest an improved scheme that uses a two-phase evaluation method that takes advantage of both the vertex and fragment processors. Since the whole deformation process takes place on the GPU and in the context of rendering, normal vectors for the vertices of the deformed geometry should be computed as well. In Section 2.4, this issue is addressed. Lastly, in Section 2.5, we show how bilinear interpolation can be used to evaluate B-spline functions of order o as efficiently as B-spline function of order $o - 1$. Section 2.5 defines the required evaluation procedure with the anticipation that future GPU would support bilinear interpolation of floating point textures.

2.1. *Evaluation of B-spline Basis Functions in the GPU*

Many algorithms exist for evaluating B-spline basis functions with arbitrary knot sequences, and can be found in standard text books [7,8]. Given a knot sequence, $\boldsymbol{\tau} = \{t_i\}$, and a parameter value, t, find i such that $t_i \leq t < t_{i+1}$. Then, the recursive relation of

$$B_i^o(t) = \frac{t - t_i}{t_{i+o-1} - t_i} B_i^{o-1}(t) + \frac{t_{i+o} - t}{t_{i+o} - t_{i+1}} B_{i+1}^{o-1}(t),$$
$$B_i^1(t) = \begin{cases} 1 & t_i \le t < t_{i+1}, \\ 0 & \text{otherwise}, \end{cases} \tag{4}$$

can be used to evaluate the i^{th}, B-spline basis function $B_i^o(t)$ of order o at t_0. Due to the locality property of B-splines basis functions, only the values of o basis functions, $B_j^o(t)$, $j = i - o + 1, \ldots, i$ should be computed. Although recursion is not readily available on contemporary GPUs, we will focus on evaluating basis functions of predefined degree, where the recursive formula can be unfolded into a direct computation.

In the case of B-spline basis functions with uniform knot sequences, fast and simple evaluation techniques do exist [9]. Assume that the knot values are selected over the integers, that is, $t_i = i$. Then, using recursive relation (4), we get,

$$\begin{aligned} B_i^o(t) &= \frac{t - i}{[i + (o-1)] - i} B_i^{o-1}(t) + \frac{[i+o] - t}{[i+o] - (i+1)} B_{i+1}^{o-1}(t) \\ &= \frac{t - i}{o - 1} B_i^{o-1}(t) + \frac{i + o - t}{o - 1} B_{i+1}^{o-1}(t). \end{aligned} \tag{5}$$

From which, one can get (see [8]),

$$B_{i+j}^o(t) = B_i^o(t - j), \tag{6}$$

implying that uniform B-spline basis functions are merely translated versions of each other. In the ensuing discussion, and unless otherwise stated, we assume uniform knot sequences.

On contemporary GPUs, the uniform B-spline basis functions could be evaluated directly for a given value t from their analytic, piecewise polynomial form. Using this method, a single precision accuracy is attained. Alternatively, one can pre-sample the basis functions, $B_i^o(t)$, on the CPU using an arbitrarily selected evaluation algorithm. These pre-computed samples would be stored on the GPU as a 32-bit floating point 1-D texture map, T_B. Specifically, for quadratic (cubic) basis functions, the values of

$$\left[B_j^{3(4)}(t)\right]_{j=i-3}^{i} = \left[(B_{i-3}^4(t),)B_{i-2}^{3(4)}(t), B_{i-1}^{3(4)}(t), B_i^{3(4)}(t)\right]$$

can be efficiently stored into the RGB(A) color channels of T_B. Using t as a texture coordinate, $\{B_j^o(t)\}_{j=i-3}^{i}$ can be approximated by sampling T_B once at t. On the other hand, one can improve the accuracy and reduce texture sizes by linearly interpolating two consecutive samples in the texture map. Since bilinear interpolation of 32-bit textures is not currently supported on contemporary GPUs, this interpolation should be implemented manually inside the shader.

In the quadratic case, the direct evaluation of $\left[B_{i-2}^3(t), B_{i-1}^3(t), B_i^3(t)\right]$ requires less then ten multiplications and additions. Using the pre-sampled table-driven approach requires one texture fetch for a nearest-neighbor value or two texture fetches and optionally, a vectorial linear interpolation (lerp). In our tests, for the quadratic case, the pre-sampled table version was shown to be slower by 20% compared to the direct polynomial evaluation method, probably due to the needed texture fetches in the vertex processor. For higher order B-spline basis functions, the pre-sampled table approach is expected to give a better relative performance.

2.2. *Single Phase FFDs Evaluation*

The most natural way to implement HFFDs is in the vertex processor, an approach denoted by us as the Single Phase (SP) approach. For every vertex $V_i \in \mathcal{O}$, the vertex processor could execute a vertex shader that would evaluate Equation (3). The only issue that needs to be resolved is how to deliver the proper parameters to the shader. As we already noted, a single evaluation of a tri-quadratic (tri-cubic) B-spline function is affected by $3 \times 3 \times 3$ ($4 \times 4 \times 4$) control points. Selecting these points should be done in a per-vertex context.

A naive approach might select the proper control points on the CPU from the control lattice of F for each vertex, V_i. Then, the control points would be transferred to the GPU along with the position of V_i in the parametric domain of F. Lastly, $F(V_i)$ could be evaluated in the vertex shader, resulting in the deformed position of V_i in the world's coordinate system. These operations take place at the vertex processor so $F(V_i)$ should also be transformed to the view space and shaded. In this case, for each vertex, the CPU must perform quite a few operations to locate the proper control points. Furthermore, a substantial amount of data will be moved to the GPU, which can put a heavy load on the graphics bus.

A better evaluation approach, which minimizes the overhead of using the CPU, would store all the control points of F on the GPU in the form of a texture map. GPUs are highly optimized for fast retrieval of data from texture maps. The only problem with this approach is that textures, until recently, were inaccessible in the vertex processor. This has changed with the introduction of vertex textures, in shader model 3.0 [10].

Using vertex textures, it is possible to store all the control points of F as a small texture map, $T_{in} = [P_{ijk}]$, which stores the xyz-position of P_{ijk} in the RGB color channels. Since floating-point volumetric textures are not currently accessible from within the vertex shader, the three-dimensional lattice of control points of the HFFDs function is laid out in sequence on a two-dimensional floating point texture maps (see Figure 2). Then, the spatial coordinates of each vertex, $V_i \in D$, are used as texture coordinates in T_{in}, retrieving all the control points that are needed for the evaluation of $F(V_i)$. Although vertex-texture queries are currently more limited than fragment-texture queries, the vertex processor of modern GPUs is still capable of about 33 million texture-fetch operations per second [10]. Unfortunately, the above evaluation scheme requires more temporary memory than available. For a tri-quadratic, $3^3 = 27$ control points must be stored in the temporary memory of the GPU, with an additional three vectors for the basis functions. In the current state-

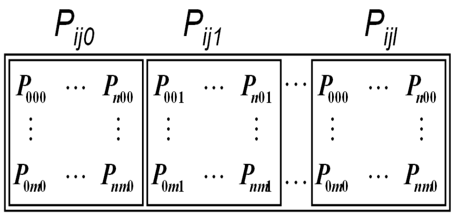


Fig. 2. A three-dimensional lattice of control points of F, laid out in sequence on a two-dimensional floating point texture map, denoted T_{in}.

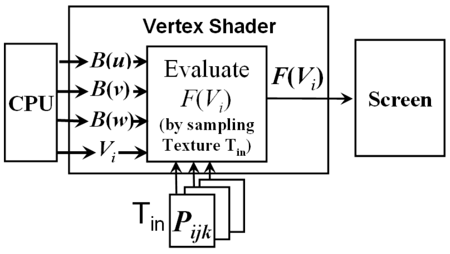


Fig. 3. The Single-Phase (SP) HFFDs algorithm that is executed solely on the vertex processor.

of-the-art systems (for example Nvidia 6800 GPU), only 32 temporary registers are available. Hence, technically, the $30 = 27 + 3$ registers could fit into the memory; in practice, however, more space is required for evaluating iterations and for holding temporary variables. One could expend this computation serially, or use a multi phase rendering algorithm at the cost of reduced performance. To alleviate this lack of memory and still retain an efficient, matrix-based evaluation scheme, the read-only registers of the GPU were used. Since read-only registers can only be written from the CPU, some computations have to be pulled back. Accordingly, the basis functions are computed once per vertex on the CPU and stored as part of the input values for the vertex shader. The above algorithm is illustrated in Figure 3, where the quadratic B-spline basis functions, $B(t) = [B_0^3(t), B_1^3(t), B_2^3(t)]$, $t \in \{u, v, w\}$, are provided as inputs to the vertex shader along with the vertex position.

The clear downside to this approach is revealed when trying to animate an object, $\mathcal{O}$, inside the parametric domain of F. In such cases, the basis functions must be recomputed for each vertex, V_i, at every animation step. Consequently, the method puts a major burden on the CPU and the graphics bus and fails to achieve interactive rates. An improved two-phase approach, which performs all the computations in the GPU, is suggested in Section 2.3.

2.3. *Two-Phase FFDs Evaluation*

To better utilize the available hardware assets in the GPU, it is natural to ask whether the deformation algorithm can also take advantage of the fragment processor. If an algorithm could be divided into two parts, the first part can be executed on the vertex processor and the second part on the fragment processor(s), exploiting the full depth of the GPU's computational pipeline. Moreover, since the fragment processor offers much better texture query support, bottlenecks due to the performance of the vertex texture should be eliminated.

The Two Phase (TP) HFFDs algorithm is shown in Figure 4. In the first phase, the deformation is computed by employing both the vertex shader and the pixel

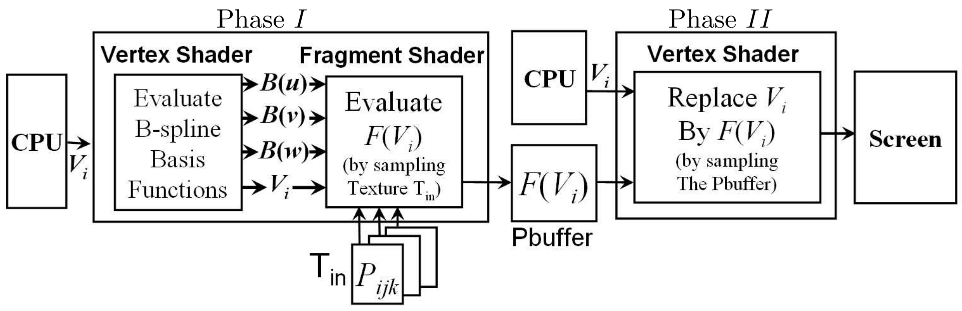


Fig. 4. The Two Phase (TP) algorithm for HFFDs. The algorithm employs both the vertex processor and the pixel processor to compute F.

shader. In the second phase, the rendering is conducted by replacing V_i by $F(V_i)$. To evaluate F, the B-spline basis functions are evaluated on the vertex processor (see Section 2.1). Toward this end, the vertex shader only has to know the position of each vertex in the parametric domain of F. Then, the fragment shader is called for each fragment, and the texture map, T_{in}, which contains P_{ijk}, the control points of F, is sampled. At this stage, the fragment shader possesses all the necessary information for evaluating Equation (3) and the deformed position of the vertex can be computed. The results of the deformation, $F(V_i)$, are rendered into an off-screen buffer. In this work, we used *pbuffers* [11]. Additionally, a more recent OpenGl extension, FrameBufferObject [12], was used. This extension is potentially faster and is much easier to use.

Nonetheless, there is one difficulty in the two-phase scheme. The vertex shader is evaluated for each vertex $V_i \in \mathcal{O}$, forming a trivial one-to-one mapping between the data and the computational kernel that processes it. The difficulty is found in the need to ensure that for each invocation of the vertex processor, a single fragment shader will be called. To circumvent this difficulty, the object is rendered as a 2D-grid of vertices onto a plane orthogonal to the viewing direction. Moreover, each vertex, V_i, is rendered inside the viewing frustum so that no vertices are clipped or z-filtered. The result is a single frame that encodes the deformed positions, $F(V_i)$, into the color channels of the pbuffer. This concludes the first phase of the algorithm. Note that this limits the number of vertices that can be handled to the number of pixels of the maximal off-screen frame buffer.

In the second phase of the algorithm, the model is actually rendered. The pbuffer, which was constructed in the first phase, is reattached as a new texture and used as the input to this phase (see Figure 4). Recall that in the first phase, a 2D-grid representation of the model was generated. Each pixel in that image corresponds to a single vertex in the model and holds its deformed position. Hence, in the second phase, the output image is sampled from the vertex shader, resulting in the deformed position $F(V_i)$ that replaces the original vertices, V_i. Finally, the model is also transformed and lit to properly render the deformed model. In our tests, the SP approach proved to be 50% slower than the TP approach. Proper rendering of the deformed model also requires the computation of the normals of the deformed geometry, a topic discussed in the next section.

2.4. *Normal Estimation*

To properly render the deformed geometry, the direction of the normal for each deformed vertex V_i, $\vec{N}_i$, should also be computed. While $\vec{N}_j$ is orthogonal to $\mathcal{O}$, this is not necessarily the case for $F(\vec{N}_i)$ and $F(\mathcal{O})$, since F is generally a non-conformal mapping. Yet, the correct normal $\tilde{N}_i$ of $F(\mathcal{O})$ at $F(V_i)$ can be approximated by mapping the complementary, tangent, space at V_i. A-priori compute two new independent vectors $\vec{T}_i^1$, $\vec{T}_i^2 \in I\!R^3$ that are orthogonal to $\vec{N}_i$, the original normal at V_i. $\vec{T}_i^1$ and $\vec{T}_i^2$ span the tangent space of $\mathcal{O}$ at V_i. $F(\vec{T}_i^1)$ and $F(\vec{T}_i^2)$ span the tangent space of the mapped geometry at $F(V_i)$, and can be approximated as $F(\vec{T}_i^j) \cong F(V_i + \vec{T}_i^j \epsilon) - F(V_i),\ j = 1, 2$ for some small $\epsilon \in I\!R^+$.

During the first phase of the deformation algorithm (see Section 2.3), two new points, $P_i^j = V_i + \epsilon T_i^j, j \in \{1, 2\}$, are rendered in addition to V_i. In the second phase, for each vertex V_i, the pbuffer is sampled three times, for $F(V_i)$, $F(P_i^1)$ and $F(P_i^2)$. Then, the direction of the normal of $F(V_i)$ is approximated as $\tilde{N}_i = \left[F(V_i) - F(P_i^1)\right] \times \left[F(V_i) - F(P_i^2)\right]$. This approximation scheme for computing normals triples the amount of work in the two rendering passes. Note that this approach also reduces the effective size of the off-screen pbuffer, and hence, the maximal size of the available models. Due to memory constraint of the SP algorithm, this method is only applicable to the TP method.

2.5. *Efficient Evaluation of B-spline Basis Functions*

For many applications, quadratic B-splines offer enough expressive power. However, there are cases where smoother deformation functions are required. In such cases, cubic B-spline functions might become desirable. When trying to use the formulation of Sections 2.2 and 2.3, the restrictions of the GPU become clear. Since there is a limited amount of read/write registers in contemporary GPUs, not all intermediate values can be stored in memory. A simple computation, similar to that in Section 2.2 for the cubic case, shows that more than seventy registers are required. In the rest of this section we present a general scheme for using the hardware capabilities to evaluate B-spline basis functions of order o in terms of B-spline basis functions of order $o - 1$. Specifically, this approach could be used to evaluate cubic trivariate B-spline functions in terms of quadratic functions, as was earlier shown in Section 2. Consider the B-spline curve, $C(t)$, of order o over knot sequence $\boldsymbol{\tau}$.

Traversing only basis functions that are non-zero for $t_i \leq t < t_{i+1}$, one gets,

$$\begin{aligned}
C(t) &= \sum_{m=i-o+1}^{i} P_m B^{o}_{m,\boldsymbol{\tau}}(t) \\
&\underset{(a)}{=} \sum_{m=i-o+1}^{i} P_m \left[\frac{t-t_m}{t_{m+o-1}-t_m} B^{o-1}_{m,\boldsymbol{\tau}}(t) + \frac{t_{m+o}-t}{t_{m+o}-t_{m+1}} B^{o-1}_{m+1,\boldsymbol{\tau}}(t) \right] \\
&= \sum_{m=i-o+1}^{i} P_m \frac{t-t_m}{t_{m+o-1}-t_m} B^{o-1}_{m,\boldsymbol{\tau}}(t) + \sum_{m=i+2-o}^{i+1} P_{m-1} \frac{t_{m+o-1}-t}{t_{m+o-1}-t_m} B^{o-1}_{m,\boldsymbol{\tau}}(t) \\
&\underset{(b)}{=} \sum_{m=i-o+2}^{i} \left[P_m \frac{t-t_m}{t_{m+o-1}-t_m} + P_{m-1} \frac{t_{m+o-1}-t}{t_{m+o-1}-t_m} \right] B^{o-1}_{m,\boldsymbol{\tau}}(t), \qquad (7)
\end{aligned}$$

where (a) is due to the recursive relation of Equation (4) and (2) is due to the locality of B-spline basis functions. Then, Equation (7) becomes

$$\begin{aligned}
C(t) &= \sum_{m=i+2-o}^{i} \left[P_{m-1}(1-\alpha) + P_m \alpha \right] B^{o-1}_{m,\boldsymbol{\tau}}(t), \quad \alpha = \frac{t-t_m}{t_{m+o-1}-t_m}, \\
&= \sum_{m=i+2-o}^{i} Q_i B^{o-1}_{m,\boldsymbol{\tau}}(t). \qquad (8)
\end{aligned}$$

This shows that the new control points Q_i are simply a linear combination of the original control points.

For B-spline basis functions with uniform knot sequences, Equation (8) can be further simplified. As in Section 2 and without loss of generality, assume the uniform knot sequences are over the integers, $t_i = i$. Then $\alpha = \frac{t-t_m}{t_{m+o-1}-t_m} = \frac{t-m}{o-1}$.

In the case of FFDs, a trivariate version of Equation (8) should be used. An evaluation matrix, similar in form to that of Equation (3), is constructed. However, the entries in this matrix are efficiently computed by bilinear texture sampling from the control point texture, T_{in}. The rest of the evaluation scheme remains the same, using Equation (3), over basis functions that are one degree smaller. Hence, evaluating an $[o \times o \times o]$ order trivariate would cost a multiplicative factor, ψ, which is the cost ratio between a bilinear and a nearest-neighbor texture sampling, more than the evaluation of an $[o-1 \times o-1 \times o-1]$ order trivariate. Since, as of today, no GPU exists that supports bilinear sampling of 32 bit floating-point textures, this method could become available only in future generation GPUs.

3. Results

The HFFDs algorithm was implemented in Cg [13], and integrated into an existing OpenGL-based viewer that is part of the Irit [14] solid modeler. The presented images were all captured in run-time. In the examples below, we used uniform tri-quadratic B-spline functions for deformation. In all cases, the TP method was used. The timing statistics were taken from a system equipped with an Nvidia6800 GPU

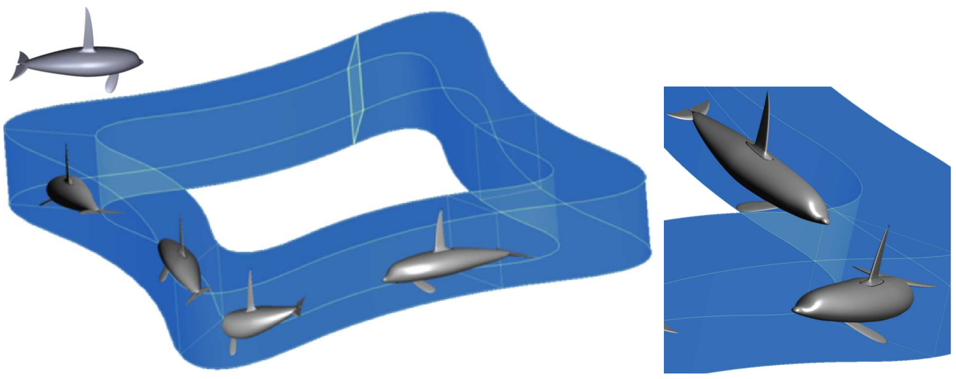


Fig. 5. Four swimming whales are animated around a periodic deformation function. Each whale model contains 28212 vertices. The non-deformed whale model is shown on the top-left. A zoom-in on the two of the whales is shown on the right.

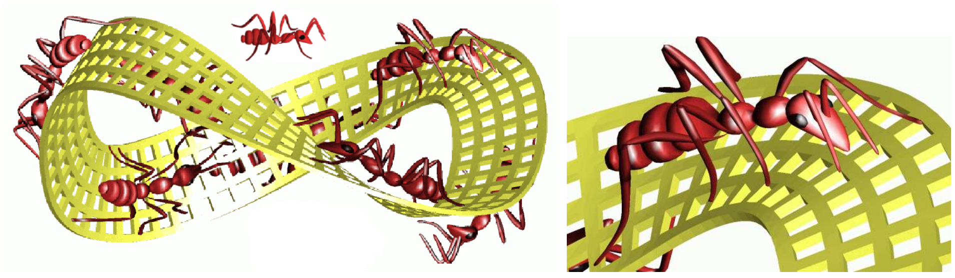


Fig. 6. A snapshot from an animation sequence of seven walking ants. The shape of each ant is deformed to fit the shape of a Möbius band. Each ant model contains 10500 vertices. The non-deformed ant model is shown on the bottom of the left hand image. A zoom-in on one ant is shown on the right. The scene is modeled after a drawing by M.C. Escher.

which possesses 4 vertex processors and 12 pixel processors. Additionally, our test system was equipped with a 4xAGP bus (instead of the recommended 8xAGP bus). All the models that are shown in the examples use simple triangle lists.

Figure 5 shows a snapshot from an animation sequence of four swimming whales. The total number of mapped vertices in the scene is 112848. We were able to achieve 7.5 frames per seconds when executing this animation. The animation is achieved by supplying the vertex shader with an animation offset in the direction of the animation for each vertex $V_i \in \mathcal{O}$ in the static whale model. On the vertex shader, this offset was added to the stored position before evaluating F.

In Figure 6, a line of seven ants is walking on a Möbius band, in the style of one of M. C. Escher's drawings. The Möbius band serves as the base surface of a trivariate B-spline function. Each ant model has 10500 vertices and the total number of vertices that are mapped through F is 73500. This example reaches 8.4 frames per seconds. In the example, an articulated animation sequence of a straight-walking ant is embedded inside the parametric domain of F. HFFDs is

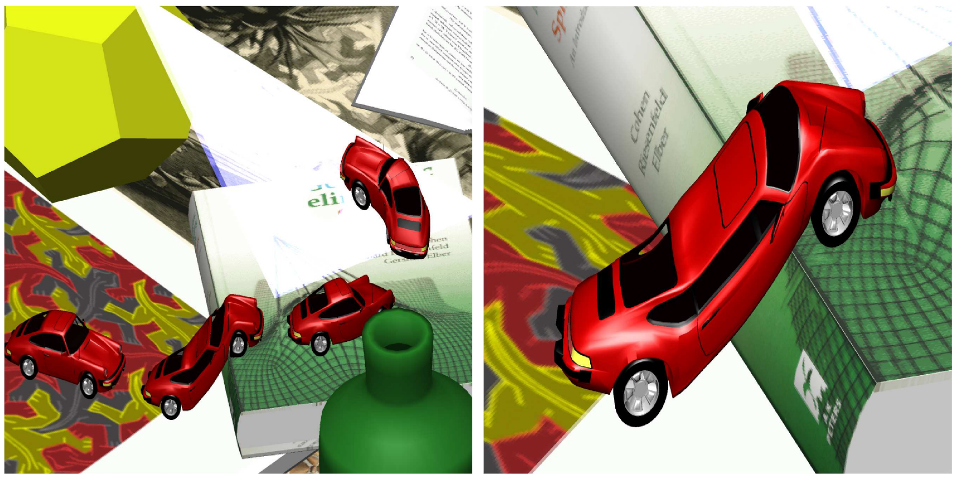


Fig. 7. On the left, a composite of four snapshots from an animation sequence of a moving Porsche. The cars are Jello-warped to drive along complex terrain. The right image shows a zoom-in on one of the cars. The Porsche model has 66000 vertices.

applied to each ant model to create an animation sequence of an articulated and deformed object, interactively. This example also shows that trivariate deformation functions can be used, in real-time, to better utilize existing straight animation sequences, adapting them to arbitrary animation routes that are defined as a single deformation function.

Figure 7(left) shows a composition of frames from an animation sequence of a deformed Porsche model. Figure 7(right) shows a zoom-in on one frame. The Porsche is Jello-warped as it drives along a path following a complex terrain. In this example, the model of a Porsche car is deformed inside a complex scene, to exemplify that HFFDs can be integrated into existing OpenGL-based applications. The Porsche model contains about 66000 vertices. This example achieved about 5.5 frames per second.

4. Conclusions and Future Directions

In this work, a method for hardware-based FFDs (HFFDs) evaluation is presented. Using the proposed approach, complex models can be deformed at interactive frame rates. Tri-quadratic B-spline functions are used as the deformation function, while we also showed how tri-cubic functions might fit into this paradigm. This method can be integrated easily into existing rendering applications. The proposed tool also offer direct access to the control points of the underlying FFDs function. By moving and dragging the control points we are able to interactively deform the existing model.

This work exemplifies two basic difficulties in implementing general purpose algorithms on GPUs. First, GPUs possess only a limited amount of temporary registers and no secondary storage. As more advanced algorithms are being ported to GPUs, the restriction on the amount of read/write memory becomes ever more problematic. In such cases, an approach that would take advantage of both the vertex processor and the fragment processor could circumvent this difficulty. To accomplish this, we proposed an approach that rendered vertices as single points on the screen, each traversing through the primitive assembly and rasterization stages of the graphics pipeline with very little overhead. Since each vertex generates a single fragment, it also generates an invocation of a single fragment shader. Using this approach, the computations can be broken up into two parts. Since there are fewer vertex processing units than fragment processing units and they operate with different performance rates, care must be taken so that the vertex processing stage does not become a bottleneck.

A second design issue that is demonstrated here is the problem of feedback. The GPU's architecture is that of a stream processor, which moves data only downstream. If an algorithm requires feedback, special considerations must be taken. The standard way feedback systems are implemented on the GPU is by writing the results of one phase of the algorithm to an off-screen buffer, here done using a pbuffer. This pbuffer is later reattached as a texture to make it available for subsequent parts of the algorithm. The downside of using pbuffers is that they are currently supported solely on Win32 platforms. A second limitation of pbuffers is found in the need to make rendering-context switches, which are rather time-consuming, to further access the pbuffer. In the future, better feedback mechanisms should be explored.

References

1. T. W. Sederberg, S. R. Parry, Free-form deformation of solid geometric models, Transactions on Graphics 20 (4) (1986) 151–160.
2. J. Griessmair, W. Purgathofer, Deformation of solids with trivariate b-splines, in: Eurograpghic 89, 1989, pp. 137–148.
3. C. Chua, U. Neumann, Hardware-accelerated free-form deformation, in: SIGGRAPH/EUROGRAPHICS Workshop On Graphics Hardware, Interlaken, Switzerland, 2000, pp. 33 – 39.
4. S. Coquillart, Extended free-form deformation: A sculpturing tool for 3d geometric modeling, in: Proceedings of the 17th Annual Conference on Computer Graphics and Interactive Techniques, Vol. 24, ACM Press, 1990, pp. 187–196.
5. M. Segal, K. Akeley, The OpenGl Graphics System: A Specification, 2nd Edition (October 2004).
6. J. Jiang, W. Luk, D. Rueckert, Fpga-based computation of free-form deformations, in: IEEE International Conference on Field-Programmable Technology, IEEE Computer Society Press, Hong Kong, China, 2002, pp. 407 – 410.
7. G. Farin, Curves and Surfaces for CAGD, 4th Edition, Academic Press, 1996.
8. E. Cohen, R. Riesenfeld, G. Elber, Geometric Modeling with Splines: An Introduction, 1st Edition, AK Peters, 2001.
9. E. Cohen, R. F. Riesenfeld, General matrix representations for bezier and b-spline curves, Computers in Industry 3 (1-2) (1980) 9–15.

10. P. Gerasimov, R. Fernando, S. Green, Shader Model 3.0, Using Vertex Texture, Nvidia Corporation, white paper Edition (2004).
11. C. Wynn, OpenGL Render-to-texture, Nvidia Corporation, white paper Edition (2004).
12. J. Juliano, J. Sandmel, EXT_framebuffer_object, OpenGL, http://oss.sgi.com/projects/ogl-sample/registry/EXT/framebuffer_object.txt (April 2005).
13. W. R. Mark, S. R. Glanville, K. Akeley, M. J. Kilgard, Cg: A system for programming graphics hardware in a c-like language, ACM Transactions on Graphics 22 (3) (2003) 896–907.
14. Irit, Irit 9.5 Solid Modeling Environment. (2006).
URL `http://www.cs.technion.ac.il/∼irit`