Volume Preserving FFD for Programmable Graphics Hardware

Stefanie Hahmann, Georges-Pierre Bonneau, Sébastien Barbier, Gershon Elber, Hans Hagen

Preprint (to appear in The Visual Computer 28, 2012)

Volume Preserving FFD for Programmable Graphics Hardware

Stefanie Hahmann*
Laboratoire Jean Kuntzmann
INRIA, University of Grenoble

Georges-Pierre Bonneau*
Laboratoire Jean Kuntzmann
INRIA, University Grenoble

Sébastien Barbier*
Laboratoire Jean Kuntzmann
INRIA, University Grenoble

Gershon Elber†
Technion, Israel

Hans Hagen‡
TU Kaiserslautern

Abstract

Free Form Deformation (FFD) is a well established technique for deforming arbitrary object shapes in space. Although more recent deformation techniques have been introduced, amongst them skeleton-based deformation and cage based deformation, the simple and versatile nature of FFD is a strong advantage, and justifies its presence in nowadays leading commercial geometric modeling and animation software systems. Since its introduction in the late 80's, many improvements have been proposed to the FFD paradigm, including control lattices of arbitrary topology, direct shape manipulation and GPU implementation. Several authors have addressed the problem of volume preserving FFD. These previous approaches either make use of expensive non-linear optimization techniques, or resort to first order approximation suitable only for small-scale deformations. In this paper we take advantage from the multi-linear nature of the volume constraint in order to derive a simple, exact and explicit solution to the problem of volume preserving FFD. Two variants of the algorithm are given, without and with direct shape manipulation. Moreover, the linearity of our solution enables to implement it efficiently on GPU.

1 Introduction

Freeform deformation is a versatile and powerful shape deformation method. Most of leading commercial geometric modeling and computer animation systems such as Maya, Softimage XSI, 3DS MAX have integrated an FFD tool. FFD is a global space deformation method. Herein, an object is embedded into a deformable space such that each point of the object has a unique parameter in the space. The object is then deformed accordingly to a deformation defined on the space. It works independently of the representation of the embedded object.

Traditionally, the deformations are defined by parametric polynomial trivariate functions parameterized over the unit cube. The control points of a Bézier or B-spline representation form a parallelepipedical lattice which serves as deformation tool. The positions of the deformed object are calculated as the image of their initial parameterization using the control points and the basis functions. While the deformation is independent of the representation of the embedded geometry, FFD is commonly used to deform polygonal models. Even very complex models with hundreds of thousands of vertices can be deformed quite intuitively by moving only a few control points.

Figure 1: Animation of an deformable object (called RedBox) flattened out on the ground. Frames 1, 25, 30, and 35 are shown. Left column: standard FFD without volume preservation. Loss of volume 28%, 51% and 64% in the frames 25,30 and 35. Right column: volume preserving FFD with volume correcting displacements of the grid points restricted to horizontal directions.

*INRIA - Laboratoire LJK
655 Avenue de l'Europe, 38330 Montbonnot, France,
e-mail:Stefanie.Hahmann@inria.fr, Georges-Pierre.Bonneau@inria.fr
†Computer Science Department, Technion, Haifa 32000, Israel
e-mail: gershon@cs.technion.ac.il
‡FB Informatik, Postfach 3049, D-67653 Kaiserslautern, Germany

Many extensions and improvements of the basic FFD method [23] have been proposed. Extensions are generally concerned with modifications of the structure of the lattice. Improvements aim to increase the realism of the deformations obtained with FFD. A first extension [5] allows a user to generate non-parallelepipedical lattices and the combination of many lattices to form arbitrary shaped spaces. However, the linear precision property gets lost, numerical techniques have to be employed for object embedding and continuity constraints are reducing the flexibility. A second extension introduces deformation methods defined on lattices of arbitrary topology [16]. The deformable space is defined by using a volumetric analogy of subdivision surfaces. This method gains in flexibility but looses in efficiency. The subdivision steps and embedding procedure are quite time consuming. Furthermore the construction of the lattice is complex.

Shape improvements for FFD seek to increase realism of deformation and animation of 3D objects either by proposing direct shape manipulation tools, by using physical-based deformation techniques, or by introducing volume preservation methods. Direct shape manipulation offers the possibility to place exactly the object points instead of manipulating FFD control points. Shape manipulation becomes thus more intuitive. [10, 11] propose to manipulate the embedded object directly by specifying source points on the object and the target points. Repositioning of the lattice control points is computed using a least-squares formulation. Hsu et al. [10] use a pseudo-inverse matrix and perform iterative computations. Hu et al. [11] add linear constraints to the least-squares formulation and derive an explicit closed form solution. The objective of physical-based animations is to increase realism by generating plausible behaviors. However, real-time animations are generally limited to small models with a thousand of control points [6].

Volume preservation is a well known principle in Computer Graphics and Animation leading to more realistic looking deformations [14]. It is a popular tool not only for FFD. Rappaport et al. [19] introduce volume preserving parametric free-form solids. The volume preservation of the lattice cells is obtained by least-squares with non-linear constraint. A non-linear optimization algorithm is employed. Convergence is not guaranteed and many time consuming iterations are necessary to approximate the given volume up to 10^{-4}. Aumann and Bechmann [1] propose a volume preserving FFD combined with position constraints for triangular models. A least-squares problem with linear constraints is solved using a pseudo-inverse matrix. However, the volume constraint is linearized by a first order Taylor approximation, which is valid only for small scale deformations. Several iterations are necessary without any guarantee of convergence. Hirota et al. [9] solve volume preserving FFD on triangular models by using non-linear minimization techniques. Convergence is not guaranteed. An extension to parametric surfaces uses multi-level optimization techniques. Both approaches are extremely time consuming.

All of these methods propose a volume correction step by recomputing the FFD lattice as close as possible to the user defined deformation. However, either the use of non-linear optimization techniques is required which is very time consuming, or the volume constraint is linearized by using a first order approximation, which limits the method to small scale deformations. Many iterations are generally needed in order to get acceptable precision, whereas no guarantee of convergence can be given.

Although CPUs have become more powerful, it is still a challenge to obtain both realism through volume preservation, and real-time FFDs for complex objects. In [21] a first step towards real-time is made by providing an implementation of FFD on programmable graphics hardware but without volume preservation.

The goal of the present paper is to propose a volume preserving FFD which exactly preserves the volume and which makes a step further towards real-time FFD with volume preservation. We achieve this goal by making the following two contributions:

- first, based on the trilinearity of the volume constraint, we propose an **explicit closed form solution** for **exact volume preservation** which optionally can include a point constraint for direct shape manipulation. No linear system has to be solved and no iterative optimization method is necessary;
- second, we provide for the first time a GPU implementation for volume preserving FFD.

2 FFD

An FFD is defined as a mapping $\overline{D} : \Omega \subset \mathbb{R}^3 \rightarrow \mathbb{R}^3$, deforming a region of $\mathbb{R}^3$ into another region of $\mathbb{R}^3$. An object which is embedded in the parameter domain Ω undergoes the same deformation. Even though $\overline{D}$ can be an arbitrary function, trivariate tensor product Bézier or B-spline functions are usually used due to their robustness and direct control via control points. The partition of unity property of the basis functions ensures linear precision. B-splines however offer more flexibility and low degree functions. We use a B-spline FFD defined as

$$\overline{D}(u,v,w) = \sum_{i=0}^{n_u}\sum_{j=0}^{n_v}\sum_{k=0}^{n_w} \boldsymbol{P}_{ijk} B_{ijk}(\boldsymbol{u}), \quad \boldsymbol{u} = (u,v,w) \in \Omega,$$

where

$$B_{ijk}(\boldsymbol{u}) := B_i(u) B_j(v) B_k(w) \tag{1}$$

and where $\boldsymbol{P}_{ijk}$ are the control points and $B_i(u)$ are the univariate B-spline basis functions of degree d relative to some knot sequence T. In the rest of the paper we replace the triple sum $\sum_{i=0}^{n_u}\sum_{j=0}^{n_v}\sum_{k=0}^{n_w}$ by the notation $\sum_{ijk}$. Let $\boldsymbol{u} = (u,v,w)$ denote the parameter value. Without loss of generality we assume the parameter domain $\Omega = [0,1]^3$ to be the unit cube. More details about B-splines can be found in standard text books [7, 4].

Let S be a polygonal surface model to be deformed by $\overline{D}$. The exact deformation of a polygonal model is approximated by applying the deformation function $\overline{D}$ only to the vertices $\boldsymbol{x}_i = (x_i, y_i, z_i)^T \in \mathbb{R}^3$ of S. S is supposed to be embedded in the parametric domain of $\overline{D}$, otherwise an affine transformation is applied to the vertices $\boldsymbol{x}_i$.

3 Volume Preserving FFD

Our goal is to conserve the volume V_{ref} embedded inside a given model, when the model is deformed by an FFD. We propose to proceed in **two steps**. First, a classical FFD given by $\overline{D}(u,v,w) = \sum_{ijk} \boldsymbol{P}_{ijk} B_i(u) B_j(v) B_k(w)$ is applied to the model. Let us denote the deformed model by $\overline{M}$. This operation may change the volume enclosed by the model, thus in general $V(\overline{M}) \neq V_{ref}$. In a second step, a *volume correction* step, the FFD grid is adjusted in order to recover the original volume. The adjustment to the FFD grid should be minimal in order to respect the deformation prescribed by the user. To this end a set of offsets $\boldsymbol{\delta}_{ijk} = (\delta^x_{ijk}, \delta^y_{ijk}, \delta^z_{ijk})^T$ is computed that will be applied to the original control points of $\overline{D}$ so that the volume of the embedded object is conserved. Let us denote the final volume preserving FFD by

$$D(u,v,w) = \sum_{ijk} (\boldsymbol{P}_{ijk} + \boldsymbol{\delta}_{ijk}) B_{ijk}(\boldsymbol{u}), \tag{2}$$

and M the resulting surface model deformed by D which satisfies the volume constraint $V(M) = V_{ref}$.

Additionally, the volume preservation part can be coupled with a *direct manipulation* mechanism [10, 11]. Direct manipulation is related to the process where the user picks a source point $\boldsymbol{S}$ on the model and moves it to a target position $\boldsymbol{T}$ in space. Then, the system computes a set of offsets that is compatible with the target position. As stated above, the requirements on the unknowns are twofold, satisfying the volume constraint and keeping the surface as close as possible to the deformed surface $\overline{M}$. Mathematically the problem states as follows:

Problem 1 Volume preservation
Compute offsets (displacements) $\boldsymbol{\delta}_{ijk}$ for the volume correcting FFD (2) such that

$$\min \sum_{ijk} \|\boldsymbol{\delta}_{ijk}\|^2 \quad \textit{subject to} \quad \Delta V(M) = 0 \tag{3}$$

where $\Delta V(M) := V(M) - V_{ref}$, see also Section 4.

Problem 2 Volume preservation and direct manipulation
Compute offsets (displacements) $\boldsymbol{\delta}_{ijk}$ for the volume correcting FFD (2) such that

$$\min \sum_{ijk} \|\boldsymbol{\delta}_{ijk}\|^2 \qquad \textit{subject to} \tag{4}$$

$$\Delta V(M) = 0 \qquad \textit{and} \tag{5}$$

$$\boldsymbol{T} = \boldsymbol{S} + \sum_{ijk} \boldsymbol{\delta}_{ijk} B_{ijk}(\boldsymbol{u}_s) \tag{6}$$

where $(\boldsymbol{u}_s)$ is the parameter value of the constraint surface point $\boldsymbol{S}$ in the parameter domain Ω of D.

Remark 1: Cost functions
Instead of minimizing squared distances $\|\boldsymbol{\delta}_{ijk}\|^2$ between FFD grid points it could also be possible to minimize the squared distance between surface mesh points before and after adjustment. But it can be easily shown that this implies to solve a linear system of equations. Our method based on the FFD grid distance however leads to an explicit solution and avoids inverting a linear system. Other quadratic cost functions could be used, such as least squares edge lengths [9] or classical linearized bending energy functionals [19]. The way to solve the problem would be analogous to the present solution. The resulting surface however would not be as close as possible to the user's defined deformation as it is our goal here.

Remark 2: Local volume preservation
Alternatively we can use weighted least-squares $\omega_{ijk}\|\boldsymbol{\delta}_{ijk}\|^2$, and use the weights ω_{ijk} in order to localize the measure of deformation. For example, a large weight ω_{ijk} tends to keep the FFD point P_{ijk} fixed. This weighted version thus enables to control where the adjustment takes place. It still leads to an explicit solution. An illustration of local volume preservation is given in Figure 7 with including comments in Section 5.2.

4 Volume computation

The oriented interior volume of a closed, orientable, piecewise triangular surface S with vertices $\boldsymbol{x}_i$ is given by

$$V(S) = \sum_{face(l,m,n)\in S} \frac{z_l + z_m + z_n}{6} \begin{vmatrix} (x_m - x_l) & (y_m - y_l) \\ (x_n - x_l) & (y_n - y_l) \end{vmatrix}, \tag{7}$$

where $face(l,m,n) = \Delta(\boldsymbol{x}_l, \boldsymbol{x}_m, \boldsymbol{x}_n)$ is a triangle of S. This trilinear expression can be derived from the volume functional for smooth surfaces, see [15]. It corresponds to the sum of the (signed) volumes of the prisms spanned by the surface triangles and their projections onto the xy-plane. The vertices $\boldsymbol{x}_s$ of the resulting volume preserving surface M are obtained by applying the FFD function D to the vertices of the initial model, i.e. $\boldsymbol{x}_s = \sum_{ijk}(\boldsymbol{P}_{ijk} + \boldsymbol{\delta}_{ijk})B_i(u_s)B_j(v_s)B_k(w_s)$, where (u_s, v_s, w_s) are the parameter values of the initial surface model in Ω. Injecting these vertices into (7) shows the dependence of the volume of M on the unknown offset vectors:

VOLUME of *M*

$$V(M) = \sum_{face(l,m,n)\in\overline{M}} \frac{\bar{z}_l + \delta_l^z + \bar{z}_m + \delta_m^z + \bar{z}_n + \delta_n^z}{6} \cdot \tag{8}$$

$$\begin{vmatrix} (\bar{x}_m - \bar{x}_l) + (\delta_m^x - \delta_l^x) & (\bar{y}_m - \bar{y}_l) + (\delta_m^y - \delta_l^y) \\ (\bar{x}_n - \bar{x}_l) + (\delta_n^x - \delta_l^x) & (\bar{y}_n - \bar{y}_l) + (\delta_n^y - \delta_l^y) \end{vmatrix}, \tag{9}$$

where

$$\delta_l^x := \textstyle\sum_{ijk} \delta_{ijk}^x B_i(u_l) B_j(v_l) B_k(w_l)$$

$$\bar{\boldsymbol{x}}_s = \textstyle\sum_{ijk} \boldsymbol{P}_{ijk} B_i(u_s) B_j(v_s) B_k(w_s).$$

The volume of M is therefore a trilinear function of the offset vector's coordinates.

5 Closed form solutions

In this section we derive an exact and closed form solution of problems 1 and 2 defined in Section 3. We avoid solving a time consuming non-linear optimization problem which is caused by the non-linear volume constraint. Instead we propose to transform the problem into a least-squares fitting problem with linear constraints. But we don't linearize the volume constraint by a first order Taylor approximation as it has been done in [1]. Instead, we use the trilinearity of the volume constraint in order to satisfy the volume constraint exactly. Furthermore, we do not solve a linear system of equations, instead we develop closed form solutions.

The volume constraint $\Delta V(M) = 0$ is a trilinear function of the unknowns δ_{ijk}^x, δ_{ijk}^y, δ_{ijk}^z, see equation (8). Thus separating the volume correcting deformation according to the axes makes it linear in each axis. We therefore set equal to zero two coordinate functions, for example δ_{ijk}^y, δ_{ijk}^z, and express the volume linearly with respect to the remaining δ_{ijk}^x, i.e. $\boldsymbol{\delta}_{ijk} = (\delta_{ijk}^x, 0, 0)$. This amounts to replace the trilinear volume constraint by a linear constraint in the optimization problems (3) and (4). In order to avoid a non-symmetric solution, we perform three successive deformations by repeating the volume correction successively according to the x, y and z-axes separately.

In order to better visualize the linearity, let us introduce the following linearized but **exact** volume formulas for the volume correcting surface M:

$$V^x(M) := \sum_{ijk} \alpha_{ijk}\ \delta_{ijk}^x + V(\overline{M}) \tag{10}$$

$$V^y(M) := \sum_{ijk} \beta_{ijk}\ \delta_{ijk}^y + V(\overline{M}) \tag{11}$$

$$V^z(M) := \sum_{ijk} \gamma_{ijk}\ \delta_{ijk}^z + V(\overline{M}) \tag{12}$$

where

$$\alpha_{ijk} := \sum_{face(l,m,n)\in\overline{M}} \frac{(B_{ijk}(\boldsymbol{u}_l) + B_{ijk}(\boldsymbol{u}_m) + B_{ijk}(\boldsymbol{u}_n))}{6} \tag{13}$$

$$\cdot \begin{vmatrix} \bar{y}_m - \bar{y}_l & \bar{y}_n - \bar{y}_l \\ \bar{z}_m - \bar{z}_l & \bar{z}_n - \bar{z}_l \end{vmatrix} \tag{14}$$

$$\beta_{ijk} := \sum_{face(l,m,n)\in\overline{M}} \frac{(B_{ijk}(\boldsymbol{u}_l) + B_{ijk}(\boldsymbol{u}_m) + B_{ijk}(\boldsymbol{u}_n))}{6} \tag{15}$$

$$\cdot \begin{vmatrix} \bar{z}_m - \bar{z}_l & \bar{z}_n - \bar{z}_l \\ \bar{x}_m - \bar{x}_l & \bar{x}_n - \bar{x}_l \end{vmatrix} \tag{16}$$

$$\gamma_{ijk} := \sum_{face(l,m,n)\in\overline{M}} \frac{(B_{ijk}(\boldsymbol{u}_l) + B_{ijk}(\boldsymbol{u}_m) + B_{ijk}(\boldsymbol{u}_n))}{6} \tag{17}$$

$$\cdot \begin{vmatrix} \bar{x}_m - \bar{x}_l & \bar{x}_n - \bar{x}_l \\ \bar{y}_m - \bar{y}_l & \bar{y}_n - \bar{y}_l \end{vmatrix} \tag{18}$$

A detailed development of the preceeding formulas can be found in Appendix A.

Problems 1 and 2 can now be re-formulated and **explicitly** solved as follows: in each step the amount of the volume will be corrected by a third with respect to one of the three axes. Therefore, we solve the problem three times by replacing the volume constraint $\Delta V(M) = 0$ successively by one of the following **linear volume constraints** $V^{x,y,z}(M) \stackrel{!}{=} V(\overline{M}) + \frac{1}{3}(V_{ref} - V(\overline{M}))$ which are equivalent to

$$\sum_{ijk} \alpha_{ijk}\ \delta^x_{ijk} = a \tag{19}$$

$$\sum_{ijk} \beta_{ijk}\ \delta^y_{ijk} = a \tag{20}$$

$$\sum_{ijk} \gamma_{ijk}\ \delta^z_{ijk} = a, \tag{21}$$

with $a = \frac{1}{3}(V_{ref} - V(\overline{M}))$. These equations are linear in δ^x_{ijk}, δ^y_{ijk}, and δ^z_{ijk} respectively. Note that after the three steps the volume will be satisfied exactly with $V(M) = V_{ref}$.

5.1 Solution of Problem 1

The first of the three optimization steps solves

$$\min \sum_{ijk} (\delta^x_{ijk})^2 \qquad \text{subject to} \tag{22}$$

$$\textit{volume constraint:} \quad \sum_{ijk} \alpha_{ijk}\ \delta^x_{ijk} = a. \tag{23}$$

The problem can be converted to a saddle point problem using a Lagrangian multiplier λ:

$$\max_{\lambda} \min_{\delta^x_{ijk}} \mathcal{L}(\delta^x_{ijk}, \lambda)$$

where

$$\mathcal{L}(\delta^x_{ijk}, \lambda) = \sum_{ijk} (\delta^x_{ijk})^2 + \lambda(\sum_{ijk} \alpha_{ijk}\ \delta^x_{ijk} - a).$$

A **closed form solution** of (22) is given by

$$\delta^x_{ijk} = \frac{a}{\sum_{rst} \alpha^2_{rst}}\ \alpha_{ijk}. \tag{24}$$

Proof:
The solution satisfies $\nabla\mathcal{L} = 0$ which is equivalent to

$$\frac{\partial \mathcal{L}}{\partial \lambda} = 0 \quad \Leftrightarrow \quad \sum_{rst} \alpha_{rst}\ \delta^x_{rst} = a \tag{25}$$

$$\frac{\partial \mathcal{L}}{\partial \delta^x_{ijk}} = 0 \quad \Leftrightarrow \quad 2\delta^x_{ijk} + \lambda\alpha_{ijk} = 0. \tag{26}$$

Inserting (26) into (25) gives the value of λ and thus the result. □

The same procedure is then repeated by replacing the linearized volume constraint in (22) by (20) and then by (21). Note that β_{ijk} of the second volume constraint (20) is computed *using the output of the first step*. And analogously for γ_{ijk}. The corresponding closed form solutions are obtained analogously. The final solutions $\boldsymbol{\delta}_{ijk}$ will precisely satisfy the volume constraint $\Delta V(M) = 0$ (to within machine precision).

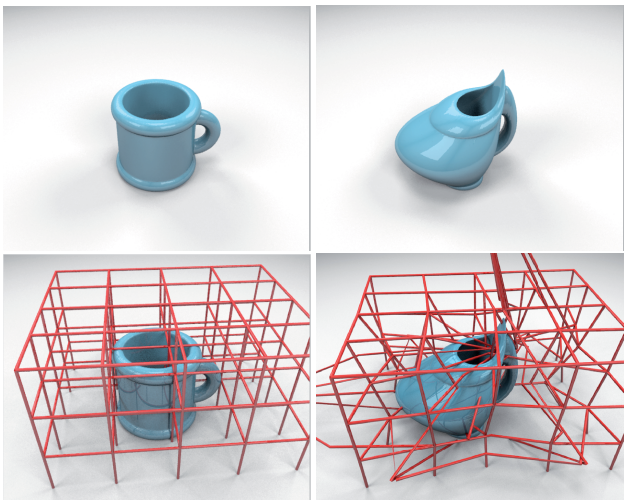

Figure 2: Sculptured cup with volume preservation.

Examples:

We applied our algorithm to various FFDs and analyzed its results in terms of design effects and visual realism. We used the fact that the linearization of the volume constraint results from applying the volume correction step only in one axis direction at the time, possibly followed by the other directions. The possibility of selecting only one or two axis directions in order to satisfy the volume constraint can have a desired design effect which we will show in the following examples.

General volume preservation is achieved by applying the volume correction successively following the x, y, and z-axis, i.e. by computing $\boldsymbol{\delta}_{ijk} = (\delta^x_{ijk}, \delta^y_{ijk}, \delta^z_{ijk})$ in three steps. Figure 2 shows such a general volume preserving deformation. On the left the original model (5.668 vertices) is shown with its initial grid. Then a specific deformation has been applied combined with a general volume correction. The result is shown in Figure 2-right.

Sometimes it might be desirable to limit the volume correction displacements $\boldsymbol{\delta}_{ijk}$ of the grid points to one or two axis directions.

A typical example is the "bouncing ball", see Figure 1. The ball when falling on the ground (e.g. in z-direction) is flattened out and simultaneously stretched horizontally (in x- and y-direction) while the inner volume is maintained constant. Thus when simulating the bouncing ball, volume correction should be limited to horizontal directions (x- and y-directions in the present case). Volume correction in vertical z-direction here would act against the natural physical behavior. Such a volume correction limited to horizontal directions can be achieved with our method by computing $\boldsymbol{\delta}_{ijk} = (\delta^x_{ijk}, \delta^y_{ijk}, 0)$ in two steps by using constraints (19) and (20) with $a = \frac{1}{2}(V_{ref} - V(\overline{M}))$ when solving problem (22). In Figure 1 four frames of a bouncing ball animation are shown, except that we replaced the geometrically simple sphere by the complex RedBox example (710.322 vertices) which has many fine details. This model as well as the cup and the screw driver are provided via the Aim@Shape[1] repository. The three deformations on the left column of Figure 1 represent a loss of volume of 28%, 51% and 64% using standard FFD. The right column shows the corresponding volume preserving deformations.

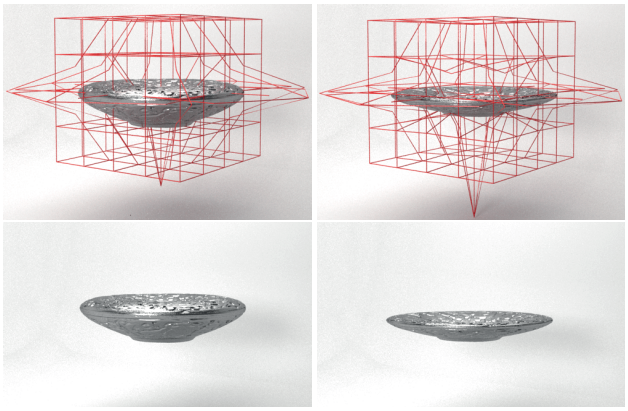

Figure 3: RedBox flattened. Left image shows the user defined FFD without volume preservation. The volume increased by 300% with respect to the initial model. Right: The flattening effect can be maintained through volume preserving FFD as well. To this end the displacement of the grid points has been limited to horizontal directions.

The next two Figures 3 and 4 used the same RedBox model and applied some FFDs where the grids have been deformed by displacing some of the grid points in order to achieve special deformations, see Figures 3,4-left. In Figure 3-left the standard FFD is shown. The object triples the volume. In Figure 3-right the volume preserving FFD is shown. It can be observed that the flattening of the model is further enforced by the volume preservation. This effect is achieved by limiting the displacements of the grid points in horizontal directions only (i.e. x and y directions here). In Figure 4-left the FFD transforms the RedBox model into a squared object with a loss of volume of 60%. This squared effect is maintained through volume preservation. In Figure 4-middle volume correction limited to horizontal displacements of the grid points is applied. In Figure 4-right volume correction limited to vertical displacements is applied. The vertical volume correction $\boldsymbol{\delta}_{ijk} = (0, 0, \delta^z_{ijk})$ is obtained by solving problem (22) only in z-direction with constraint (21) where $a = (V_{ref} - V(\overline{M}))$.

Further special FFDs are applied to a screw driver in Figure 5. The original model is shown on the left, the two volume preserving deformations can be seen on the right.

[1] `http://shapes.aim-at-shape.net`

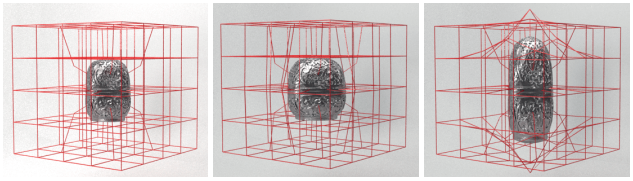

Figure 4: RedBox squared. Left: standard FFD with 40% loss of volume. Middle: Volume preserving FFD with volume correction displacements only in horizontal directions $\boldsymbol{\delta}_{ijk} = (\delta^x_{ijk}, \delta^y_{ijk}, 0)$. Right: Volume correction displacements only in vertical direction $\boldsymbol{\delta}_{ijk} = (0, 0, \delta^z_{ijk})$.

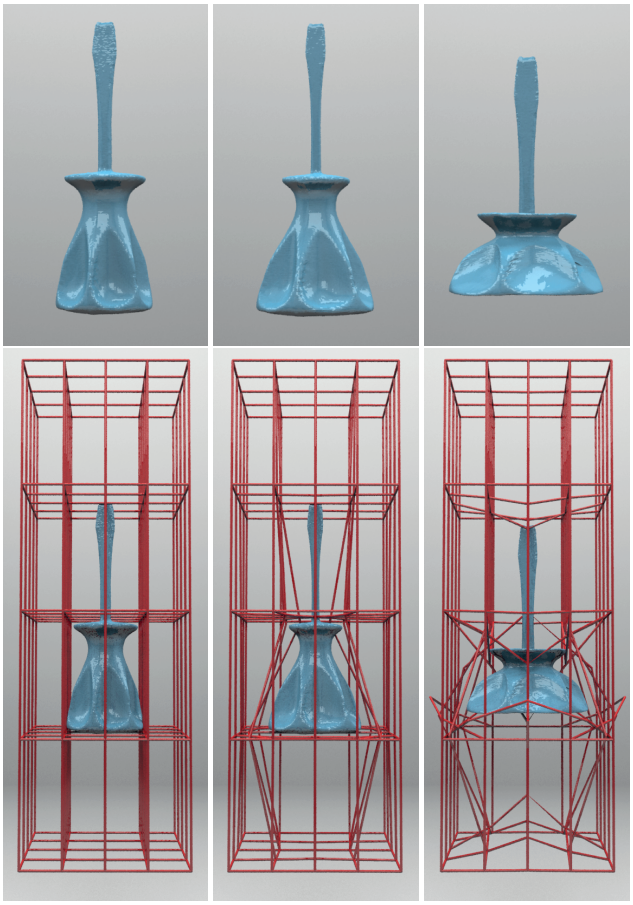

Figure 5: Screwdriver redesigned. Left: initial model with 27.152 vertices. Middle: volume preserving deformation with volume correcting displacements restricted to horizontal directions $\boldsymbol{\delta}_{ijk} = (\delta^x_{ijk}, \delta^y_{ijk}, 0)$. Right: Correction displacements restricted to vertical direction $\boldsymbol{\delta}_{ijk} = (0, 0, \delta^z_{ijk})$.

Continuously twisting an object is a classical FFD. We applied this deformation in Figure 6 to a cube model (6146 vertices) with and without volume preservation. In the first row the volume decreases during the animation. At the last frame 66% of the initial volume is lost. In the second row volume preservation is enforced along the vertical axis only, i.e. $\delta^x_{ijk} = \delta^y_{ijk} = 0$. The third row shows the same animation with exact volume preservation using

only the x- and y-directions, i.e. by keeping $\delta^z_{ijk} = 0$. The advantage is that the upper and lower face of the cube remain flat during animation. This might be a desired effect, which can not be achieved with general volume preservation.

5.2 Solution of Problem 2

In addition to the volume constraint, a position constraint is added in Problem 2. Once again, a closed form solution can be derived by decomposing the problem accordingly to a correction with respect to the x, y and z-axes. In a first step we solve

$$\min \sum_{ijk} (\delta^x_{ijk})^2 \qquad \text{subject to} \tag{27}$$

$$\textit{volume constraint:} \quad \sum_{ijk} \alpha_{ijk}\ \delta^x_{ijk} = a \tag{28}$$

$$\textit{position constraint:} \quad T^x = S^x + \sum_{ijk} \delta^x_{ijk} B_{ijk}(\boldsymbol{u}_s) \tag{29}$$

where (u_s, v_s, w_s) is the parameter of $\boldsymbol{S}$ in the parameter domain of D. Using Lagrangian multipliers λ and μ we solve the minmax problem:

$$\max_{\lambda,\mu} \min_{\delta^x_{ijk}} \mathcal{L}(\delta^x_{ijk}, \lambda, \mu)$$

with the Lagrangian function

$$\mathcal{L}(\delta^x_{ijk}, \lambda, \mu) = \sum_{ijk} (\delta^x_{ijk})^2 + \lambda((T^x - S^x) - \sum_{ijk} \delta^x_{ijk} B_{ijk}(\boldsymbol{u}_s)) \tag{30}$$

$$+ \mu(\sum_{ijk} \alpha_{ijk}\ \delta^x_{ijk} - a). \tag{31}$$

A **closed form solution** of (27) is given by

$$\delta^x_{ijk} = \lambda \frac{B_{ijk}(\boldsymbol{u}_s)}{2} - \mu \frac{\alpha_{ijk}}{2} \tag{32}$$

where

$$\lambda = \frac{2(T^x - S^x)(\sum_{ijk} \alpha^2_{ijk}) - 2a \sum_{ijk} \alpha_{ijk} B_{ijk}(\boldsymbol{u}_s)}{(\sum_{ijk} \alpha^2_{ijk})(\sum_{ijk} B^2_{ijk}(\boldsymbol{u}_s)) - (\sum_{ijk} \alpha_{ijk} B_{ijk}(\boldsymbol{u}_s))^2}$$

and

$$\mu = \frac{2(T^x - S^x)(\sum_{ijk} \alpha_{ijk} B_{ijk}(\boldsymbol{u}_s)) - 2a \sum_{ijk} B^2_{ijk}(\boldsymbol{u}_s)}{(\sum_{ijk} \alpha^2_{ijk})(\sum_{ijk} B^2_{ijk}(\boldsymbol{u}_s)) - (\sum_{ijk} \alpha_{ijk} B_{ijk}(\boldsymbol{u}_s))^2}.$$

Proof:
The solution satisfies $\nabla \mathcal{L} = 0$ which is equivalent to the following three equations:

$$\frac{\partial \mathcal{L}}{\partial \lambda} = 0 \quad \Leftrightarrow \quad T^x = S^x + \sum_{rst} \delta^x_{rst} B_{rst}(\boldsymbol{u}_s) \tag{33}$$

$$\frac{\partial \mathcal{L}}{\partial \mu} = 0 \quad \Leftrightarrow \quad \sum_{rst} \alpha_{rst}\ \delta^x_{rst} = a \tag{34}$$

$$\frac{\partial \mathcal{L}}{\partial \delta^x_{ijk}} = 0 \quad \Leftrightarrow \quad 2\delta^x_{ijk} - \lambda B_{ijk}(\boldsymbol{u}_s) + \mu \alpha_{ijk} = 0. \tag{35}$$

δ^x_{ijk} from equation (35) is inserted into (33) and (34). One obtains a linear system of two equations in two unknowns λ and μ. An explicit solution of the system is calculated and inserted into (35). □

We then repeat this step by replacing the linearized volume constraint in (27) by (20) and then by (21) and by applying the position constraint to the y and z-coordinate. The final solutions $\boldsymbol{\delta}_{ijk}$ will satisfy exactly the volume constraint $\Delta V(M) = 0$ and the position constraint $\boldsymbol{T} = \boldsymbol{S}$.

Example:

Figure 7 illustrates two effects: direct shape manipulation and local volume preservation. The top left picture shows the original armadillo model (52.971 vertices). In the top middle picture, the tip of the ear is constraint to a position chosen by the user, but no volume correction is applied. Two undesired effects occur: first the inner volume increases, second the head is deformed by the FFD. We successfully solve both unwanted behaviors by using a local volume correction, as explained in Section 3, Remark 2. In order to localize the deformation along the ear, we use large weights ω_{ijk} for FFD points $\boldsymbol{P}_{ijk}$ far away from the constraint point. In practice we set in this example the weights for each FFD point $\boldsymbol{P}_{ijk}$ as an exponential function of the distance between $\boldsymbol{P}_{ijk}$ and the constraint point at the tip of the ear.

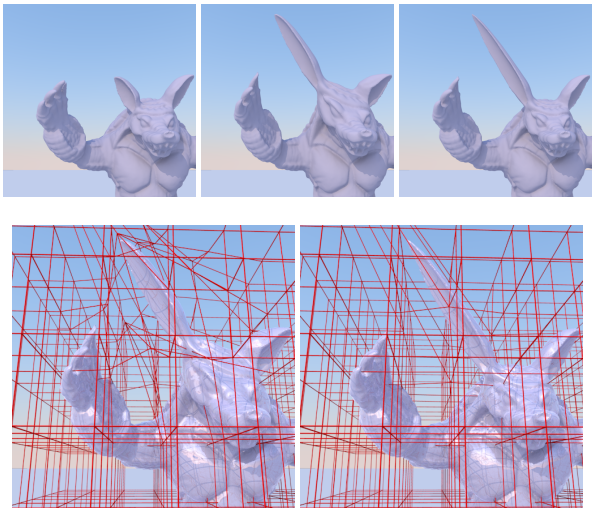

Figure 7: Direct shape manipulation with one specified position. Top left: original model. Top middle: the position of the tip of the ear is constrained, without volume preservation. Top right: same position constraint, with a local volume preservation. Bottom left (resp. bottom right): same as top middle (resp. top right), with the FFD grid in red.

Conclusion: It has been illustrated with many examples the possibility offered by our method to compute volume correction displacements of grid points only in horizontal or only in vertical direction. This may be a benefit for FFD where a privileged direction can be figured out. In other cases the general volume correction has to be applied.

We want however draw attention to the main advantage of our method : it presents an explicit solution to volume preserving FFDs and thus allows for an efficient GPU implementation, as explained in the next section.

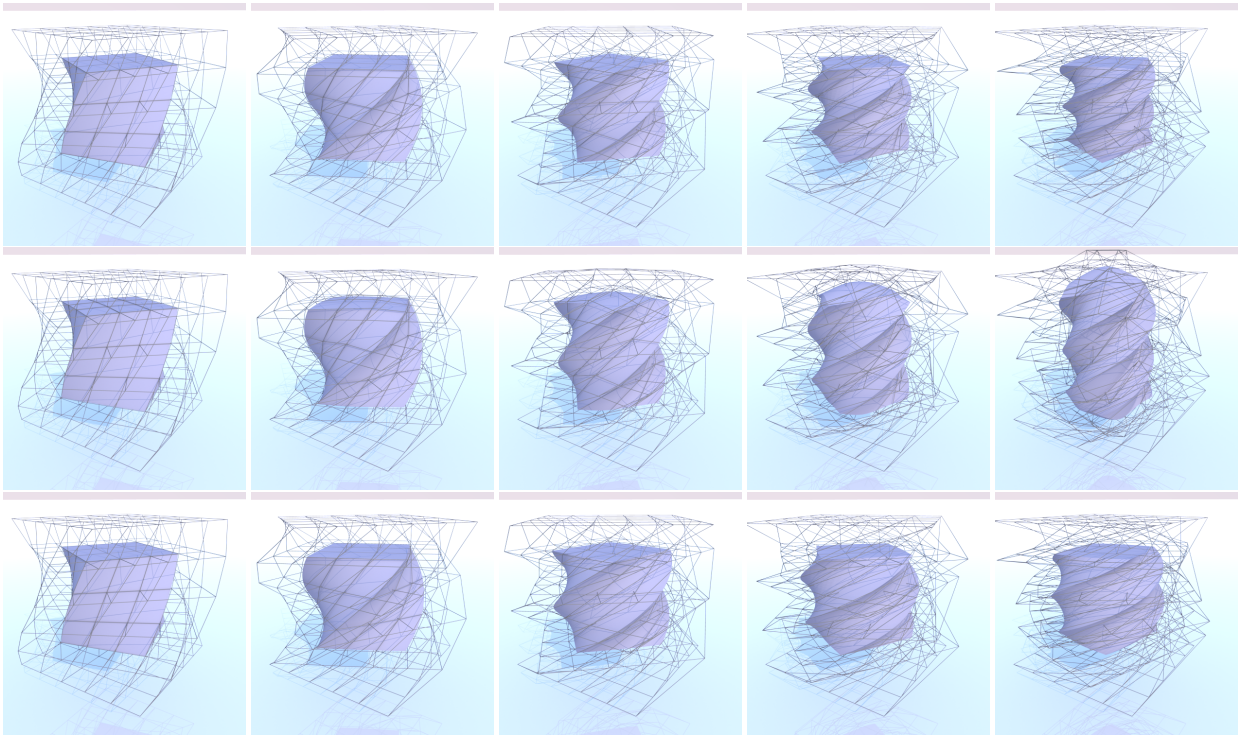

Figure 6: Animation of Twisting cube. First row is without volume preservation. At the last frame 66% of volume is lost. Second row with volume preservation displacement are limited to horizontal directions $\boldsymbol{\delta}_{ijk} = (\delta^x_{ijk}, \delta^y_{ijk}, 0)$. Third row with volume preservation limited to vertical direction $\boldsymbol{\delta}_{ijk} = (0, 0, \delta^z_{ijk})$.

6 GPU accelerated volume preserving FFD

Most existing surface deformation techniques, including FFD [21], skinning [20], physically-based deformations [18, 13, 8], deformation displacement maps [22], cage-based deformations [3], have been implemented on the GPU. Because the technique is not adapted to parallelism, or due to memory limitations, some of these deformation methods have a GPU implementation that is less efficient or less general than the CPU implementation ([20, 17, 12]). In [21] it has been shown that FFD can be accelerated significantly using programmable graphics hardware, but without volume preservation. In the present section we will focus on how to perform our volume preserving FFD fully on GPU.

In contrast to previous works on volume preserving FFD our method has the following advantages:

1. the volume is precisely preserved (to within machine precision),
2. the method is linear (non-linear optimization techniques are not needed),
3. an explicit solution is provided (no linear system has to be solved).

These properties turn out to be essential for an efficient GPU implementation.

In section 3 we have described the framework in which our volume preservation procedure is applied. An FFD grid given as input is adjusted in such a way that the volume inside the resulting mesh M has a given value V_{ref}. In our current implementation the input FFD grid is given either by user interaction or by some animation technique (we have implemented twist, bend, and simple physical-based deformation of the FFD-grid). The overall pipeline of the volume preserving FFD grid adjustment is given in Figure 8.

The following four steps are implemented on the graphics card:

- the computation of the current volume (7),
- the computation of the coefficients $\alpha_{ijk}, \beta_{ijk}, \gamma_{ijk}$ (13-17),
- the computation of the offsets δ_{ijk} by (24) or (32) and the update of the FFD grid according to these offsets,
- the computation of the FFD deformation (1).

The three first steps are implemented using CUDA kernels and the last one using a vertex shader. Context switches are done during the pipeline without new GPU memory allocation. The final rendering is performed using GLSL shaders. We detail more precisely these four steps in the following subsections.

6.1 Volume Computation Kernels

The current volume is obtained using two CUDA kernels directly from the equation (7). This formula can be decomposed into two successive steps: the computation of

- the terms $\frac{z_l+z_m+z_n}{6} \begin{vmatrix} (x_m - x_l)\ (y_m - y_l) \\ (x_n - x_l)\ (y_n - y_l) \end{vmatrix}$ (volume below each face),
- their sum.

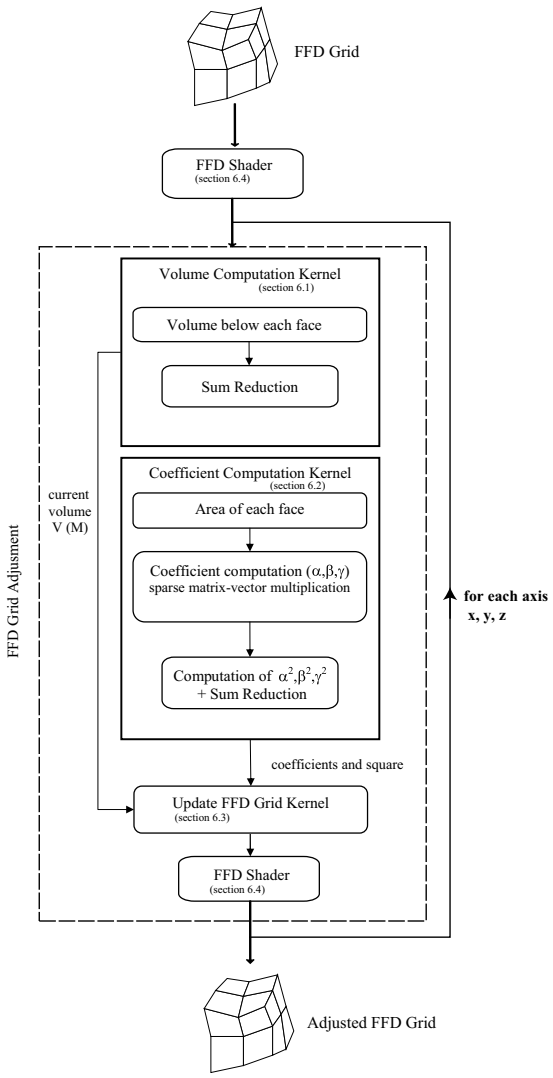

Figure 8: GPU Pipeline of our volume preserving FFD.

The first step is performed for each face where each face is considered independently as a thread. An array of size F is generated, where F is the number of faces. The resulting sum is computed using a *parallel scan* [24]. The current volume is then recorded into the GPU memory.

6.2 Coefficient Computation Kernels

The computation of the volume preservation coefficients given in equations (13-17) is performed in three steps: computation of

- the terms $\begin{vmatrix} \bar{y}_m - \bar{y}_l & \bar{y}_n - \bar{y}_l \\ \bar{z}_m - \bar{z}_l & \bar{z}_n - \bar{z}_l \end{vmatrix}$
  (resp. $\begin{vmatrix} \bar{z}_m - \bar{z}_l & \bar{z}_n - \bar{z}_l \\ \bar{x}_m - \bar{x}_l & \bar{x}_n - \bar{x}_l \end{vmatrix}$ or $\begin{vmatrix} \bar{x}_m - \bar{x}_l & \bar{x}_n - \bar{x}_l \\ \bar{y}_m - \bar{y}_l & \bar{y}_n - \bar{y}_l \end{vmatrix}$),
- the coefficients α_{ijk} (resp. β_{ijk} or γ_{ijk}),
- sum of squares $\sum_{rst} \alpha_{rst}^2$ (resp. $\sum_{rst} \beta_{rst}^2$ or $\sum_{rst} \gamma_{rst}^2$).

To compute the coefficients α_{ijk}, equation (13) is rewritten for efficiency. Indeed, implementing directly the sum as in formula (13) would require the computation of a parallel scan for each coefficient, which is prohibitively time-consuming. Instead, we compute all coefficients α_{ijk} in one step using the following sparse matrix-vector multiplication:

$$\mathbf{A} = B.\mathbf{V}$$

where

$$B_{(N\times F)} = \begin{bmatrix} & \vdots & \\ \cdots & \frac{\left(B_{ijk}(\boldsymbol{u}_l)+B_{ijk}(\boldsymbol{u}_m)+B_{ijk}(\boldsymbol{u}_n)\right)}{6} & \cdots \\ & \vdots & \end{bmatrix}$$

$$\mathbf{A}_{(N)} = \begin{bmatrix} \vdots \\ \alpha_{ijk} \\ \vdots \end{bmatrix}, \mathbf{V}_{(F)} = \begin{bmatrix} \vdots & \\ \bar{y}_m - \bar{y}_l & \bar{y}_n - \bar{y}_l \\ \bar{z}_m - \bar{z}_l & \bar{z}_n - \bar{z}_l \\ \vdots & \end{bmatrix}.$$

B is a sparse rectangular matrix of size $(N \times F)$ where $N = n_u n_v n_w$ denotes the FFD grid size, see Section 2, and F is the number of mesh faces of S. $\mathbf{V}$ is a vector of size F. $\mathbf{A}$ is the resulting (N)-vector containing the α_{ijk} coefficients.

Note that B is a sparse-matrix due to the local support property of the B-Spline basis functions [6]. B is precomputed once and loaded into the GPU memory in a sparse matrix format. Uniform basis functions can be evaluated efficiently directly from their analytic, piecewise polynomial form [7, 4]:

$$\begin{bmatrix} B_0(t) \\ B_1(t) \\ B_2(t) \\ B_3(t) \end{bmatrix} = \frac{1}{6} \begin{bmatrix} 1 & -3 & 3 & -1 \\ 4 & 0 & -6 & 3 \\ 1 & 3 & 3 & -3 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 \\ t \\ t^2 \\ t^3 \end{bmatrix}.$$

V is computed on-the-fly with a CUDA kernel applied to each face, Then a sparse-matrix-vector multiplication is applied following [2] in order to compute all coefficients in parallel.

The third step computes the square of the coefficients using a parallel-scan algorithm [24].

The coefficients α_{ijk} and their sum square $\sum_{rst} \alpha_{rst}^2$ are stored in the GPU memory.

6.3 FFD grid update kernel

The current volume $V(M)$, the coefficients α_{ijk} and their sum square $\sum_{rst} \alpha_{rst}^2$ as well as the reference volume V_{ref} are already computed at this step. A CUDA kernel is launched for each vertex of the FFD grid in order to compute and add the offsets given by (24) or (32). Notice that in our implementation, the FFD grid is stored as a 2D texture to efficiently compute the mesh transformation. The texture is consequently transformed into a CUDA array using pixel buffer object and is then remapped as texture at the end of the kernel execution.

6.4 FFD deformation shader

The FFD deformation (1) is implemented in a vertex shader, as first proposed in [21]. Contrary to [21], the parameters $\boldsymbol{u}_s$ of the mesh vertices do not change in our framework. Therefore we precompute the B-Spline coefficients $B_{ijk}(\boldsymbol{u}_s)$, and we stored them as a buffer object. The new mesh vertices are stored into a transform feedback buffer and can be used for the final rendering.

7 Experimental results

The benefits of volume preservation for getting more plausible deformations have been pointed out by many authors and need not be proven anymore. We therefore focus on experiments on the performance estimation of our volume correction step onto current graphics hardware. All models have been put into a FFD grid of size $N = n_u n_v n_w$ where $N = 6^3$ and $N = 10^3$ has been chosen for establishing the tables below. The same FFD deformation is applied to all models. The statistics we report on in this section don't depend on the particular FFD applied, but only on the grid size and the size of the model. We therefore have chosen publicly available models[2] with known sizes, but we don't show the FFD we applied in order to compute the performances. We performed a series of experiments with models of varying size on a PC Intel, 2.13 GHz with 3.5 GB Ram equipped with a GForce 8800 GTS, 512 MB with 128 Cuda cores. The computation time we measure only includes the time needed for a volume preservation in one privileged direction, i.e. the time needed to compute the $\delta_{ijk}^{x,y,z}$. The time needed for computation of the FFD is computed separately. The quantities $B_{ijk}(\boldsymbol{u}_s)$ are pre-computed for all vertices $\boldsymbol{x}_s$ of the input model since they stay fix when several succeeding deformations are applied to the same model.
For the sake of completeness, and because a sequential CPU implementation might be useful on mobile terminals without GPU, we first briefly study the sequential complexity of our algorithm.

Sequential complexity:
The complexity to solve problems 1 and 2 (see Section 3) is the same. We give the details for problem 1. First the volume of the deformed mesh $V(\overline{M})$ must be computed. According to Section 4 this costs $O(F)$. Then the coefficients $\alpha_{ijk}, \beta_{ijk}$ and γ_{ijk} have to be computed, using equations (13), (15) and (17). In a CPU implementation this can be done efficiently by traversing the faces of the mesh, and for each face by updating the contribution of this face to the coefficients $\alpha_{ijk}, \beta_{ijk}$ and γ_{ijk}. Due to the compact support of the B-Spline functions, one face contributes to a constant number of coefficients $\alpha_{ijk}, \beta_{ijk}$ and γ_{ijk}. Therefore the cost of computing the $\alpha's$, $\beta's$ and $\gamma's$ is $O(F)$. The squared sum of these coefficients must be computed. This costs $O(N)$. Finally, the FFD grid points must be updated using the values of δ_{ijk} in (24). This costs $O(N)$. The total complexity of a sequential implementation of volume preserving FFD is therefore $O(F + N)$.

Parallel complexity:
Note that in order to allow for parallelization, the GPU implementation imposes to compute all offsets $\boldsymbol{\delta}_{ijk}$ independently from each other (see Sect. 6.2). It is not possible in a GPU implementation to traverse the faces in parallel and to update the contribution of this face to different $\alpha's$, $\beta's$ and $\gamma's$. Here a matrix-vector multiplication is used. However, the matrix of size $(N \times F)$ is sparse and the sparse matrix encoding ensures that the number of basic operations for such a matrix-vector multiplication is linearly proportional to the number of non-zero elements in the matrix. Furthermore, for our particular application it can be shown that the number of non-zero elements essentially depends of the number of mesh faces F and not on the FFD grid size N. In fact, the grid nodes corresponds to the knots on which the B-spline basis functions are defined. Each mesh vertex lies in the support of exactly 64 cubic B-spline basis functions. An element (i, j) in the matrix B is non-zero, if one of the vertices of the face j lies in the support of one basis function corresponding to line i in the matrix. If the three vertices of a face j lie in the same FFD cell, there are exactly 64 non-zero elements in the corresponding column. This is the case for almost all faces in the mesh, for reasonable size N of the FFD grid. This is confirmed by Table 1. This table reports the exact number of non-zero elements in matrix B for all test models. It can be observed that the number of non-zero elements in the $N = 6^3$ grid and in the $N = 10^3$ grid is very similar and approximatively equal to $64 \times F$ for each test model.
The cost of the matrix-vector multiplication applied to compute the $\alpha's$, $\beta's$ and $\gamma's$ is thus $O(F)$. The squared sum of these coefficients and the FFD grid updating costs $O(N)$. The total complexity of $O(F + N)$ for the GPU version is thus the same as for the sequential CPU implementation.

Efficiency:
In our GPU implementation, the sparse-matrix-vector multiplication (SpMV) (see section 6.2) is the most expensive step. Since the matrix B is sparse with variable number of non zeros per row but without any particular sparsity pattern, we used the *compressed sparse row* CSR-format for matrix representation. It explicitly stores column indices and nonzero values in arrays. A third array of row pointers takes the CSR representation. We mapped the SpMV to the GPU using the CSR implementation proposed by Bell and Garland [2]. Our matrix B has the particularity that the number of rows is limited and much smaller than the number of columns ($N << F$). The comparison of several SpMV kernels in [2] shows that efficiency decreases for non-square matrices but that the CSR kernel performs best for these unstructured non-square matrices. In our case however, increasing the number of rows, i.e. the grid size, from 6^3 to 10^3 e.g. in order to equilibrate the matrix sizes may reduce the computation time for the matrix-vector multiplication, but it is compensated by increasing the cost for the remaining operations of order $O(N)$. Thus finally, we didn't observe any influence of the grid size on the computation time of our volume preserving FFD kernel and therefore chose $N = 10^3$ for all timings.

model	**# faces** F	**# non zeros** (grid size 10^3)	**# non zeros** (grid size 6^3)
Sphere	5,120	322,598 (6.3%)	301,916 (27,3%)
Porsche	10,474	680,098 (6.5%)	635,637 (28,1%)
Horse	39,698	2,306,680 (5.8%)	2,304,791 (26,9%)
Bunny	69,451	4,026,316 (5.8%)	4,006,614 (26,7%)
BallJoint	274,120	15,716,692 (5.7%)	15,649,260 (26,4%)
HandSkel	654,666	37,662,714 (5.8%)	37,474,881 (26,5%)
Dragon	871,414	49,929,232 (5.7%)	49,674,401 (26,4%)
Happy	1,087,716	62,388,900 (5.7%)	62,479,049 (26,6%)

Table 1: Sparsity of the matrix B of size $(N \times F)$. The number of non zeros in the matrix is compared to two different FFD grid sizes: $N = 10^3$ and $N = 6^3$.

Table 2 reports the model statistics and computation times in milliseconds. We measured the times needed for computing one FFD and a volume preservation in one privileged direction on CPU and on GPU. The times needed for computing an FFD is negligible. We computed in the last column the factor which tells how many times the GPU volume preservation is faster than a volume preservation performed on CPU. For example, volume preservation for the HandSkel model is 6.5 times faster on GPU than on CPU. Figure 9 compares the computation time for both implementations as linear functions of the number of mesh vertices. This plot also illustrates the linear complexity for both CPU and GPU implementations.

Our CPU algorithm can manage interactive rates until $50,000$

[2] porsche : www-roc.inria.fr/gamma/gamma.php
happy, bunny : graphics.stanford.edu/data/3Dscanrep
horse : shapes.aim-at-shape.net
handskel, dragon : www.cc.gatech.edu/projects/large_models
balljoint : www.cs.caltech.edu/~njlitke/meshes
sphere : icosahedron, subdivided 4 times, projected on unit sphere.

model	# vertices	GPU FFD	CPU VOL	GPU VOL	factor
Sphere	2,562	0.02	6	10	0.6
Porsche	5,247	0.05	12	12	1.0
Horse	19,851	0.23	46	16	2.9
Bunny	31,947	0.34	96	19	5.1
Armadillo	52,971	0.45	154	27	5.7
BallJoint	137,062	1.84	383	59	6.5
HandSkel	327,323	2.66	749	116	6.5
Dragon	437,645	3.56	1,206	156	7.7

Table 2: Timings (in milliseconds). Column "GPU FFD" shows the time to compute the FFD on GPU, without volume correction. Column "CPU VOL" (resp. "GPU VOL") indicates the time to adjust the FFD grid for volume correction, on CPU (resp. GPU). Column "factor" shows the ratio between GPU and CPU volume correction timings. We use a PC Intel, 2.13GHz, 3.5 GB RAM equipped with an NVIDIA GeForce 8800 GTS, 512 MB with 128 Cuda cores. Grid size is $N = 10^3$ for all tests.

vertices (7 fps for a volume preserving FFD). Using the GPU implementation, more than $400,000$ vertices are deformed with volume preservation at the same frame rate.

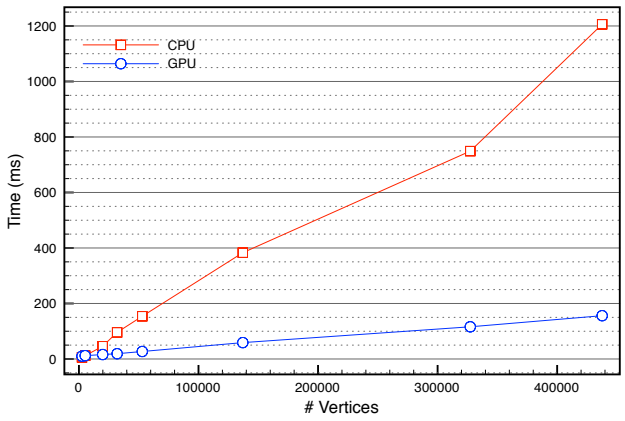

Figure 9: Computation time of a volume preserving FFD as a function of the number of mesh vertices. The red (resp. blue) line corresponds to the CPU (resp. GPU) implementation.

Volume preserving FFD in real-time (20 fps) for large meshes is thus still a challenge, even though this GPU implementation improves the computation time with respect to an analogous CPU implementation and provides interactive frame rates even for large models. Different directions for further research in order to reduce the computational cost will be discussed in the next section.

8 Conclusion and future work

We have presented a novel algorithm for computing volume preserving FFD exactly and proposed an implementation on programmable graphics hardware. We derived an exact and explicit solution to the problem of volume preserving FFD including the possibility of direct shape manipulation. We further demonstrated that an efficient implementation on the GPU is possible thanks to the explicit formulas we derived.

It turns out that most models can be deformed with exact volume preservation in real-time (20 fps), while large models can still be deformed at interactive frame rates (7 fps). The gain with respect to standard CPU implementation is thus important and justifies a GPU implementation. As with every other algorithm, our GPU volume preserving FFD has its limitations. Even though the gain in computation time is respectable, the method is limited by the performance of the matrix-vector multiplication on the GPU. In fact, a recent study of Bell and Garland [2] states this problem in case of our general unstructured matrices with a great disparity between number of rows and columns. This disparity could be reduced by adapting the algorithm to locally correct the volume, e.g. only in regions where the loss of volume is the most important. In this case the number of mesh faces involved would decrease and so the number of columns in the matrix. We would expect the efficiency of the matrix-vector multiplication to increase. How to detect these regions and how to apply the volume correction locally should be approached in the future.

Another direction to be considered for reducing the cost of the method is approximation and multiresolution. In the case where an approximate volume correction is sufficient, one could create a coarse approximation of the model and apply the method to this approximation. The resulting volume correcting FFD would thus approximately preserve the volume of the fine mesh.

Further extensions are possible which are concerned with general free-form deformations and thus apply to volume preserving FFD as well : First, one could improve visual quality of the deformation by adaptively subdividing the triangles following the deformation. Indeed, a large triangle will stay flat even if the deformation bent it and should be tesselated adaptively. Second, a negative Jacobian in the FFD function would hint on the possibility of creating self intersections in the mapped model (and negative volume). One could detect a negative Jacobian by computing the Jacobian function of the mapping as a trivariate spline (on the GPU) only to examine its zeros. If the Jacobian has only positive coefficients clearly there is no self-intersection. Otherwise one can still do a more closer examination (like insert a certain number of knots in the suspected area and reexamine). Doing it on the fly could create an FFD system that prevents self-intersections in the deformed mesh.

Acknowledgements

This work was partially supported by the DFG (IRTG 1131, INST 248/72-1).

References

[1] Fabrice Aubert and Dominique Bechmann. Volume-preserving space deformation. *Computers and Graphics*, 21(5):625–639, 1997.

[2] Nathan Bell and Michael Garland. Efficient sparse matrix-vector multiplication on cuda. Technical report, NVIDIA Technical Report NVR-2008-004, December 2008.

[3] Mirela Ben-Chen, Ofir Weber, and Craig Gotsman. Variational harmonic maps for space deformation. *ACM Trans. Graph.*, 28(3):1–11, 2009.

[4] Eliane Cohen, Richard Riesenfeld, and Gershon Elber. *Geometric Modeling with Splines: An Introduction*. AK Peters, 2001.

[5] Sabine Coquillart. Extended free-form deformation: a sculpturing tool for 3d geometric modeling. In *SIGGRAPH '90: Proceedings of the 17th annual conference on Computer graphics and interactive techniques*, pages 187–196, New York, NY, USA, 1990. ACM.

[6] Petros Faloutsos, Michiel Van De Panne, and Demetri Terzopoulos. Dynamic free-form deformations for animation synthesis. *IEEE Transactions on Visualization and Computer Graphics*, 3:201–214, 1997.

[7] Gerald Farin. *Curves and Surfaces for Computer Aided Geometric Design*. Academic Press, New York, 5th edition, 2002.

[8] Nolan Goodnight, Cliff Woolley, Gregory Lewin, David Luebke, and Greg Humphreys. A multigrid solver for boundary value problems using programmable graphics hardware. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, pages 102–111. Eurographics Association Aire-la-Ville, Switzerland, Switzerland, 2003.

[9] Gentaro Hirota, Renee Maheshwari, and Ming C. Lin. Fast volume-preserving free form deformation using multi-level optimization. In *SMA '99: Proceedings of the fifth ACM symposium on Solid modeling and applications*, pages 234–245, New York, NY, USA, 1999. ACM Press.

[10] William M. Hsu, John F. Hughes, and Henry Kaufman. Direct manipulation of free-form deformations. In *SIGGRAPH '92: Proceedings of the 19th annual conference on Computer graphics and interactive techniques*, pages 177–184, New York, NY, USA, 1992. ACM.

[11] Shi-Min Hu, Hui Zhang, Chiew-Lan Tai, and Jia-Guang Sun. Direct manipulation of ffd: Efficient explicit solutions and decomposible multiple point constraints. *The Visual Computers*, 17(6):370–379, 2001.

[12] Ladislav Kavan, Steven Collins, Jiří Žára, and Carol O'Sullivan. Skinning with dual quaternions. In *Symposium on Interactive 3D Graphics*, pages 39–46, 2007.

[13] Jens Krüger and Rüdiger Westermann. Linear algebra operators for GPU implementation of numerical algorithms. *ACM Transactions on Graphics (TOG)*, 22(3):908–916, 2003.

[14] John Lasseter. Principles of traditional animation applied to 3d computer animation. *SIGGRAPH'87 Computer Graphics*, pages 35–44, 1987.

[15] Sheue-Ling Lien and James T. Kajiya. A symbolic method for calculating the integral properties of arbitrary nonconvex polyhedra. *IEEE Computer Graphics and Applications*, 4(9), October 1984.

[16] Ron MacCracken and Kenneth I. Joy. Free-form deformations with lattices of arbitrary topology. In *SIGGRAPH '96: Proceedings of the 23rd annual conference on Computer graphics and interactive techniques*, pages 181–188, New York, NY, USA, 1996. ACM.

[17] Martin Marinov, Mario Botsch, and Leif Kobbelt. GPU-based multiresolution deformation using approximate normal field reconstruction. *Journal of Graphics, GPU, & Game Tools*, 12(1):27–46, 2007.

[18] A. Nealen, M. Müller, R. Keiser, E. Boxerman, and M. Carlson. Physically Based Deformable Models in Computer Graphics. *Computer Graphics Forum*, 25(4):809–836, 2005.

[19] Ari Rappoport, Alla Sheffer, and Michel Bercovier. Volume-preserving free-form solids. *IEEE Transactions on Visualization and Computer Graphics*, 2(1):19–27, 1996.

[20] Taehyun Rhee, J Lewis, and Ulrich Neumann. Real-time weighted pose-space deformation on the GPU. *Computer Graphics Forum*, 25(3), 2006.

[21] Sagi Schein and Gershon Elber. Real-time freedom deformation using programmable hardware. *International Journal of Shape Modeling*, 12(2):179–192, 2006.

[22] Sagi Schein, Eran Karpen, and Gershon Elber. Real-time geometric deformation displacement maps using programmable hardware. *The Visual Computer*, 21(8-10):791–800, 2005.

[23] Thomas W. Sederberg and Scott R. Parry. Free-form deformation of solid geometric models. *SIGGRAPH Comput. Graph.*, 20(4):151–160, 1986.

[24] Shubhabrata Sengupta, Mark Harris, and Michael Garland. Efficient parallel scan algorithms for GPUs. Technical report, NVIDIA Technical Report NVR-2008-003, December 2008.

Appendix A

The volume formula in equation (7) is obtained by computing for mesh faces the prism between the face and its projection following the z-axis onto the $x-y$-plane. The volume of the model is then the sum of the signed volumes of all prisms. The same volume is obtained by projecting the faces to the $x-z$-plane or the $y-z$-plane. Therefore we have the following three identical formulas for the volume of a triangular mesh whether the projection is done following the x-, y-, or z-axis:

$$V(S) = \sum_{face(l,m,n)\in S} \frac{x_l + x_m + x_n}{6} \begin{vmatrix} (y_m - y_l) & (z_m - z_l) \\ (y_n - y_l) & (z_n - z_l) \end{vmatrix} \tag{36}$$

$$\tag{37}$$

$$= \sum_{face(l,m,n)\in S} \frac{y_l + y_m + y_n}{6} \begin{vmatrix} (z_m - z_l) & (x_m - x_l) \\ (z_n - z_l) & (x_n - x_l) \end{vmatrix} \tag{38}$$

$$\tag{39}$$

$$= \sum_{face(l,m,n)\in S} \frac{z_l + z_m + z_n}{6} \begin{vmatrix} (x_m - x_l) & (y_m - y_l) \\ (x_n - x_l) & (y_n - y_l) \end{vmatrix}, \tag{40}$$

where $face(l,m,n) = \Delta(\boldsymbol{x}_l, \boldsymbol{x}_m, \boldsymbol{x}_n)$ is a triangle of M. Remember that the deformed model M is obtained by applying a volume preserving FFD to the vertices of the original model, using equation (2). Thus the volume $V(M)$ can also be expressed in terms of the deformed control points $\bar{\boldsymbol{x}}_s$ and the volume correcting

terms $\boldsymbol{\delta}_{ijk}$ as follows, see also (8):

$$
\begin{aligned}
V(M) \quad &= \textstyle\sum_{face(l,m,n)\in\overline{M}} \frac{\bar{x}_l+\delta^x_l+\bar{x}_m+\delta^x_m+\bar{x}_n+\delta^x_n}{6} \\
&\cdot \begin{vmatrix} (\bar{y}_m-\bar{y}_l)+(\delta^y_m-\delta^y_l) & (\bar{z}_m-\bar{z}_l)+(\delta^z_m-\delta^z_l) \\ (\bar{y}_n-\bar{y}_l)+(\delta^y_n-\delta^y_l) & (\bar{z}_n-\bar{z}_l)+(\delta^z_n-\delta^z_l) \end{vmatrix} \\
&= \textstyle\sum_{face(l,m,n)\in\overline{M}} \frac{\bar{y}_l+\delta^y_l+\bar{y}_m+\delta^y_m+\bar{y}_n+\delta^y_n}{6} \\
&\cdot \begin{vmatrix} (\bar{z}_m-\bar{z}_l)+(\delta^z_m-\delta^z_l) & (\bar{x}_m-\bar{x}_l)+(\delta^x_m-\delta^x_l) \\ (\bar{z}_n-\bar{z}_l)+(\delta^z_n-\delta^z_l) & (\bar{x}_n-\bar{x}_l)+(\delta^x_n-\delta^x_l) \end{vmatrix} \\
&= \textstyle\sum_{face(l,m,n)\in\overline{M}} \frac{\bar{z}_l+\delta^z_l+\bar{z}_m+\delta^z_m+\bar{z}_n+\delta^z_n}{6} \\
&\cdot \begin{vmatrix} (\bar{x}_m-\bar{x}_l)+(\delta^x_m-\delta^x_l) & (\bar{y}_m-\bar{y}_l)+(\delta^y_m-\delta^y_l) \\ (\bar{x}_n-\bar{x}_l)+(\delta^x_n-\delta^x_l) & (\bar{y}_n-\bar{y}_l)+(\delta^y_n-\delta^y_l) \end{vmatrix}
\end{aligned}
$$

with δ^x_{ijk}, δ^y_{ijk}, δ^z_{ijk} as unknowns. $V^x(M)$ is the linearized volume function of M, where δ^y_{ijk} **and** δ^z_{ijk} **are set to zero**. Therefore $V^x(M)$ is a linear function in the unknowns δ^x_{ijk}. In Section 5, equation (10) we have denoted the linearized volume formula $V^x(M)$ as follows:

$$
\begin{aligned}
V^x(M) &= V(\bar{X}+\delta X, \bar{Y}, \bar{Z}) && (41)\\
&= \sum_{ijk} \alpha_{ijk}\delta^x_{ijk} + V(\overline{M}). && (42)
\end{aligned}
$$

Let us now compute the coefficients α_{ijk}:

$$
\begin{aligned}
V^x(M) &= V(\bar{X}+\delta X, \bar{Y}, \bar{Z}) && (43)\\
&= \sum_{face(l,m,n)\in\overline{M}} \frac{(\bar{x}_l+\delta^x_l)+(\bar{x}_m+\delta^x_m)+(\bar{x}_n+\delta^x_n)}{6} && (44)\\
&\quad \cdot \begin{vmatrix} (\bar{y}_m-\bar{y}_l) & (\bar{y}_n-\bar{y}_l) \\ (\bar{z}_m-\bar{z}_l) & (\bar{z}_n-\bar{z}_l) \end{vmatrix} && (45)\\
& && (46)\\
&= V(\overline{M}) \quad + \sum_{face(l,m,n)\in\overline{M}} \frac{\delta^x_l+\delta^x_m+\delta^x_n}{6} && (47)\\
&\quad \cdot \begin{vmatrix} (\bar{y}_m-\bar{y}_l) & (\bar{y}_n-\bar{y}_l) \\ (\bar{z}_m-\bar{z}_l) & (\bar{z}_n-\bar{z}_l) \end{vmatrix} && (48)\\
& && (49)
\end{aligned}
$$

Replace now $\delta^x_l = \sum_{ijk} \delta^x_{ijk} B_{ijk}(\boldsymbol{u}_l)$ and δ^y_m, δ^z_n analogously:

$$
\begin{aligned}
V^x(M) = & \sum_{face(l,m,n)\in\overline{M}} \sum_{ijk} \frac{\delta^x_{ijk}B_{ijk}(\boldsymbol{u}_l)+\delta^x_{ijk}B_{ijk}(\boldsymbol{u}_m)+\delta^x_{ijk}B_{ijk}(\boldsymbol{u} && (50)\\
& \cdot \begin{vmatrix} (\bar{y}_m-\bar{y}_l) & (\bar{y}_n-\bar{y}_l) \\ (\bar{z}_m-\bar{z}_l) & (\bar{z}_n-\bar{z}_l) \end{vmatrix} + V(\overline{M}) && (51)\\
& && (52)\\
= & \sum_{ijk} \delta^x_{ijk} \Bigg[\sum_{face(l,m,n)\in\overline{M}} \frac{B_{ijk}(\boldsymbol{u}_l)+B_{ijk}(\boldsymbol{u}_m)+B_{ijk}(\boldsymbol{u}_n)}{6} \begin{vmatrix} (\bar{y}_m-\bar{y}_l) \\ (\bar{z}_m-\bar{z}_l) \end{vmatrix} && (53)\\
& && (54)\\
& + V(\overline{M}) && (55)
\end{aligned}
$$

The expression in square brackets is denoted by α_{ijk}. β_{ijk} (resp. γ_{ijk}) in equation (11) (resp. equation (12)) are computed analogously by setting equal zero $\delta^x_{ijk}, \delta^z_{ijk}$ (resp. $\delta^x_{ijk}, \delta^y_{ijk}$).