Vis Comput (2012) 28:585–595
DOI 10.1007/s00371-012-0705-0

ORIGINAL ARTICLE

Eigen deformation of 3D models

Tamal K. Dey · Pawas Ranjan · Yusu Wang

Published online: 18 April 2012

Abstract Recent advances in mesh deformations have been dominated by two techniques: one uses an intermediate structure like a cage which transfers the user intended moves to the mesh, the other lets the user to impart the moves to the mesh directly. The former one lets the user deform the model in real-time and also preserve the shape with sophisticated techniques like Green Coordinates. The direct techniques on the other hand free the user from the burden of creating an appropriate cage though they take more computing time to solve larger non-linear optimizations. It would be ideal to develop a cage-free technique that provides real-time deformation while respecting the local geometry. Using a simple eigen-framework, we devise such a technique. Our framework creates an *implicit* skeleton automatically. The user only specifies the motion in a simple and intuitive manner, and our algorithm computes a deformation whose quality is similar to that of the cage-based scheme with Green Coordinates.

Keywords Laplace operator · Eigenvectors · Eigenspace · Deformation · Animation

1 Introduction

The creation of deformed models from an existing one is a quintessential task in animations and geometric modeling. A user availing such a system would like to have the flexibility in controlling the deformation in real-time while preserving the isometry. In recent years, considerable progress has been made to meet these goals and a number of approaches have been suggested.

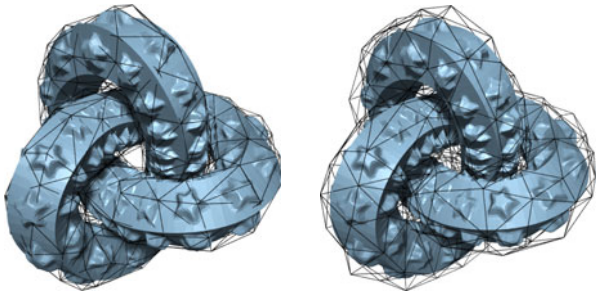
Some of the earliest techniques for mesh deformation involved using skeletons [33]. A user typically creates a skeletal shape which is bound to the mesh. The mesh is then deformed by deforming the skeleton and transforming the changes back to the mesh. This approach puts a burden on the user to create an appropriate skeleton and bind the mesh to it. Later work [1, 9, 32] sought to reduce this burden by automatically creating a skeleton. Automatic generation of good skeletons and accurate transformation of deformations from skeleton to mesh remain challenging until today.

Later approaches replaced the skeletons with a sparse cage surrounding the mesh and then controlled the deformation through the movement of the cage. The use of a cage is akin to the concept of control polyhedron that is used for free-form deformations. The authors in [27] introduced the concept of control polyhedron and others refined it later [18, 22]. It is well recognized that a control polyhedron does not provide sufficient flexibility to deform meshes with complicated topology and geometry [15, 21]. More recent techniques increase this flexibility by introducing sophisticated coordinate functions that bind the cage to the mesh. In general, each vertex of the mesh is associated with weights called *coordinates* for each vertex of the cage. This allows the mesh vertices to be represented as a linear combination of the vertices of the cage. Cage based techniques vary in how the weights of the vertices are computed. Early attempts include extending the notion of barycentric coordinates to polyhedra [16, 23, 31]. More recently, Mean Value Coordinates [10, 11, 19], Harmonic [15], and Green [21] Coordinates have been proposed for the purpose. The au-

T.K. Dey · P. Ranjan (✉) · Y. Wang
Department of Computer Science and Engineering, The Ohio State University, Columbus, OH, USA
e-mail: ranjan@cse.ohio-state.edu

T.K. Dey
e-mail: tamaldey@cse.ohio-state.edu

Y. Wang
e-mail: yusu@cse.ohio-state.edu



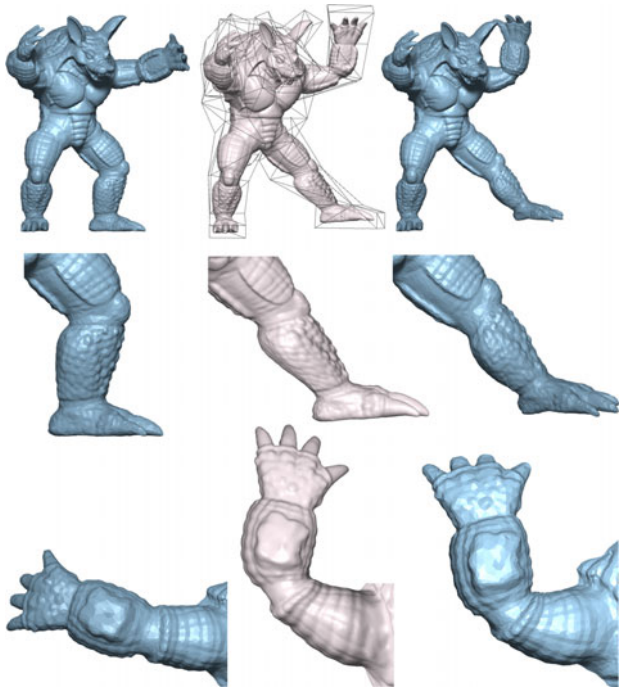
(a) cage intersects with mesh (b) self-intersecting cage

Fig. 1 Creating correct cages for meshes

thors in [21] pointed out that the Mean Value and Harmonic Coordinates do not necessarily preserve shapes though they provide affine invariant deformations. They overcome this difficulty by providing a real-time deformation tool that preserves shapes. Recently, in [14], Sorkine et al. use biharmonic coordinates to integrate cages and skeletons under a single framework. This allows the user to use different types of techniques simultaneously based on the result desired. Also, in [3], Ben-Chen et al. use a small set of control points on the original mesh to guide the deformation of the cage. Nevertheless, the limitation of creating pseudostructures like cages and skeletons by users still persists.

Creating pseudostructures, especially cages, can be time-consuming and tricky, as Fig. 1 illustrates. The cage on the left, for example, fails to envelop the mesh correctly. The cage on the right envelops the entire mesh, but has self-intersections leading to incorrect calculation of coordinates. It falls upon the user to manually move the cage vertices to rectify the cage, which can become time-consuming. A user typically spends more time creating a good cage, than deforming the mesh. The state-of-the art would be enhanced if one can have a tool that has the capabilities of Green Coordinates but without the need for the cage. Our approach is geared toward that. Figure 2 illustrates the point by showing how our method produces similar quality deformations as Green Coordinates but without any cage.

There are other approaches that impart the deformation directly to the surface mesh and thus eliminate the need for intermediate structures like cages or skeletons; see, e.g., [4, 5, 13, 29, 30, 35]. These techniques usually optimize an energy function tied to the deformation and user control to achieve high quality deformations. However, they either require nonlinear solvers or multiple iterations of linear solvers to compute new vertex positions, making them slower for large meshes. Also see [6] for a survey on various deformation techniques that use the Laplacian operator to formulate the energy.



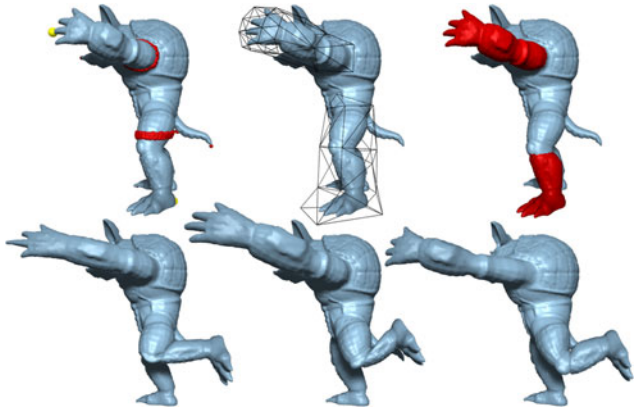
(a) Original Mesh (b) Green Coordinates (c) Our Method

Fig. 2 Comparing with green coordinates

1.1 Our work

We introduce a novel approach that allows the user to apply the deformation directly to the mesh but without solving any nonlinear system, and thus improving both time and numerical accuracy. The method uses a skeleton but without explicitly constructing one. It computes the eigenvectors of the Laplace–Beltrami operator to provide a low frequency harmonic functional basis which helps creating an *implicit* skeleton. The skeleton is a high-level abstraction of the shape of the mesh, lacking small features and details. It deforms this skeleton by computing a new set of eigen coefficients. These coefficients are solutions of a *linear* system which can be computed in real-time. Finally, it adds the details back to the skeleton to get the deformed mesh. We point out that unlike other skeleton-based approaches our implicit skeleton is simply the original mesh whose vertex coordinates are derived from a truncated set of eigenvectors.

Our eigen-framework retains the advantages of both the cage-based and cage-less approaches. Figures 2, 10, and 13 illustrate this point. Our deformation software is easy to use and efficient. We are also able to handle both isometric as well as nonisometric deformations like stretching gracefully, as shown in Fig. 3. More examples can be found in the video submitted with this paper.



(a)ARAP (b) HC (c) Our method

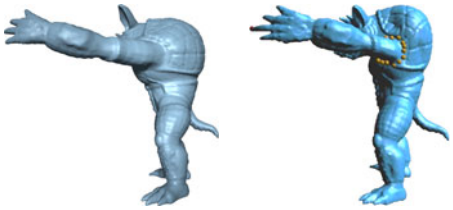
Fig. 3 Comparisons when we stretch the arm and bend the leg of the armadillo. Note that our method handles stretching better than as–rigid-as-possible (ARAP), and extreme bending better than harmonic coordinates (HC)

1.2 Comparison with previous work on spectral deformation

Recently, Rong et al. [25, 26] proposed a deformation framework using the eigenvectors of the Laplace operator. Although this seems similar to our approach, the reasons why we use eigenvectors are different. In particular, Rong et al. perform as-rigid-as-possible [29] deformations by trying to preserve the Laplace operator. However, they use the eigenvectors to change the problem domain from spatial to spectral, thereby reducing the size of the optimization problem to the number of eigenvectors used. Usually, they need at least 100 eigenvectors, and more eigenvectors make their final deformation look better.

Our method, on the other hand, just needs a functional basis of low frequency harmonic functions in which the meshes are represented. We use low frequency eigenvectors in order to get a smooth fit for a target deformation since our goal is to guarantee a *smoothly varying deformation* rather than *isometry*. Hence, we can guide deformations by using as few as 8 eigenvectors. Furthermore, since we do not try to preserve the Laplace operator, and hence isometry, we can handle stretching better than [25, 26], as Fig. 4 illustrates.

Also, [25, 26] use deformation transfer based techniques to recover details, and sometimes produce artifacts in the deformed mesh, as shown in Fig. 12. We develop a more sophisticated iterative technique to recover the details with greater accuracy. Finally, in [25, 26], the positions of the constrained vertices need to be changed each time the user wish to change the scope of the deformation, increasing users' burden to specify the intended deformation.



(a) Our Method (b) SMD

Fig. 4 Stretching the arm of the armadillo. Note that spectral mesh deformation (SMD), causes the entire mesh to deform in order to preserve the mesh volume

2 Eigen-framework

2.1 Laplace–Beltrami operator

Consider a smooth, compact surface M isometrically embedded in $\mathbb{R}^3$. Given a twice differentiable function $f : \mathsf{M} \to \mathbb{R}$, the Laplace-Beltrami operator Δ_{M} of f is defined as the divergence of the gradient of f.

The Laplace–Beltrami operator has many useful properties and has been widely used in many geometric processing applications; see [20, 34] for surveys on Laplacian mesh processing applications. For example, it is well known that the Laplace operator uniquely decides the intrinsic geometry of the input manifold M. Hence, isometric manifolds share the same Laplacian, which makes the Laplace operator a natural tool to capture or describe isometric deformation. Indeed, this idea has been used to build local coordinates for mesh editing and deformation to help produce as-rigid-as possible type of deformation [28, 35]. The eigenfunctions of the Laplace operator form a natural basis for square integrable functions defined on M. Analogous to Fourier harmonics for functions on a circle, Laplacian eigenfunctions with lower eigenvalues correspond to low frequency modes, while those with higher eigenvalues correspond to high frequency modes that describe the details of the input manifold M.

In our problem, the input is a triangular mesh approximating a hidden surface M. In such case, we need a discrete version of the Laplace operator computed from this mesh. Several choices are available in the literature [2, 7, 12, 23, 24]. In this paper, we use the mesh-Laplacian developed in [2], although other discretizations of the Laplace operator should also be fine. Both the mesh-Laplace operator itself and its eigenvalues have been shown to converge to those of the hidden manifold as the mesh approximates the manifold better [2, 8]. See [6] for a discussion of the effects that the various discretizations have on Laplacian based surface deformation techniques.

2.2 Eigenskeleton

Given the Laplace operator Δ_{M} of an input manifold, let $\phi_1, \phi_2, \ldots$ denote the eigenfunctions of Δ_{M}. These eigen-

functions form a basis for $\mathcal{L}^2(\mathsf{M})$, the family of square integrable functions on M. Hence, we can re-write any function $f \in \mathcal{L}^2(\mathsf{M})$ as $f = \sum_{i=1}^{\infty} \alpha^i \phi_i$, where $\alpha^i = \langle f, \phi_i \rangle$ and $\langle \cdot, \cdot \rangle$ is the inner product in the functional space $\mathcal{L}^2(\mathsf{M})$. Under this view, the function f can be considered as a vector $\alpha = [\alpha^1, \alpha^2, \ldots]$ in the infinite-dimensional *eigenspace* spanned by the Laplacian eigenfunctions.

Now, consider the *coordinate functions* (f_x, f_y, f_z) defined on M whose values at each point are simply the x-, y-, and z-coordinate values of the point, respectively. By rewriting these coordinate functions, we can represent a surface by three vectors $(\alpha_x, \alpha_y, \alpha_z)$ in its eigenspace. We call these the *coordinate weights* of M. The embedding of a manifold is fully decided by its coordinate weights once the eigenfunctions are given.

Finally, since higher eigenfunctions have higher frequencies and hence capture smaller details, we can truncate the number of eigenfunctions (i.e., use only the top few coordinate weights) for reconstructing the surface to get varying levels of detail.

Eigenskeleton Given a surface mesh, also denoted by M, with n vertices, we compute the eigenvectors of the mesh-Laplacian computed from M, denoted still by $\phi_1, \ldots, \phi_n$. We now restrict ourselves to the first few, say $m < n$, eigenvectors of the shape. This gives us a higher level abstraction of the surface that captures its coarser features. Specifically, let $P = \{p_1, p_2, \ldots, p_n\}$ be the set of points reconstructed from the vertex set $V = \{v_1, v_2, \ldots, v_n\}$ of M using only the first m eigenvectors $\phi_1, \phi_2, \ldots, \phi_m$ of the mesh-Laplacian. That is, if

$$\hat{f}_x = \sum_{i=1}^{m} \alpha_x^i \phi_i, \qquad \hat{f}_y = \sum_{i=1}^{m} \alpha_y^i \phi_i, \qquad \hat{f}_z = \sum_{i=1}^{m} \alpha_z^i \phi_i$$

then $p_i = \{\hat{f}_x(v_i), \hat{f}_y(v_i), \hat{f}_z(v_i)\}$ for $i = 1, \ldots, n$. Consider the mesh $\mathsf{K} = \mathsf{K}_m$ with vertices p_i and the connectivity same as that of M. We call this mesh the *eigenskeleton*[1] of M. For different values of m, the eigenskeleton K_m abstracts the input surface M at different levels of detail.

3 Algorithm

Our algorithm will compute the target configuration by deforming the eigenskeleton. In particular, the eigenskeleton K_m is decided by the $3m$ coordinate weights $\alpha_x^1, \ldots, \alpha_x^m$; $\alpha_y^1, \ldots, \alpha_y^m$ and $\alpha_z^1, \ldots, \alpha_z^m$. We will deform the eigenskeleton by computing a new set of coordinate weights by solving only a linear system. Since the number of eigenvectors used is typically much smaller than the number of vertices involved in deformation, a solution can be obtained in real-time. We will see later that, other than being efficient, the use of coordinate weights also has the advantage that the deformation tends to be smooth across the entire shape. Since the new eigenskeleton lacks smaller features, we design a novel and effective algorithm to add back details using the one-to-one correspondence between the vertices of the eigenskeleton and the original mesh. The high level framework for our algorithm is described in Algorithm 1. Next, we describe each step in detail.

Algorithm 1: Deformation framework

```
Input  : Input mesh M
Output: Deformed mesh M*
begin
    Compute eigenskeleton K for M
    While (user initiates deformation) {
        Step 1: Interpret user-specified deformation and
                compute a coarse target configuration K̃
        Step 2: Obtain K*, a smooth approximation to K̃
        Step 3: Add shape details to obtain M*
    }
```

3.1 Step 1: Coarse guess-target configuration

Our software uses a standard and simple interface for the user to specify the intended target configuration. First, the user selects a mesh region that he wishes to deform. We call it the *region of interest*, R, and let $V_R \subseteq V$ denote the set of vertices in this region. Next, the user specifies the type of transformation desired for the region of interest, which can be either a translation-type or a rotation-type. The user then indicates the target configuration by simply dragging some point, say $\mathbf{v} \in V_R$, to its target position $\tilde{\mathbf{v}}$.

From the type of transformation combined with the position of $\mathbf{v}$ and $\tilde{\mathbf{v}}$, our algorithm computes either a translational vector $\mathbf{t} = \tilde{\mathbf{v}} - \mathbf{v}$ if the desired transformation is a translation-type, or a rotational pivot $\mathbf{p}$ and a rotation matrix $\mathbf{r}$ if the desired transformation is a rotation-type. We then compute a coarse target configuration $\tilde{\mathsf{K}}$ for the eigenskeleton K using the following simple procedure: For all points $v_i \notin V_R$, the target position for the corresponding point p_i in the eigenskeleton is simply $\tilde{p}_i = p_i$. For each point $v_i \in V_R$, if the type of transformation is translation, then the target position is $\tilde{p}_i = p_i + \mathbf{t}$. If the type of transformation is rotation, then the target position is $\tilde{p}_i = \mathbf{r}(p_i - \mathbf{p}) + \mathbf{p}$.

In other words, we simply cut the region of interest and apply to it the target transformation indicated by the user, while the rest of the shape remains intact. Such an initial guess of target configuration is of course rather unsatisfactory. In fact, the deformation is not even continuous

[1] The concept of eigenskeletons is not new and has been used for mesh compression in [17]. For more applications, please refer to the survey papers [20, 34].

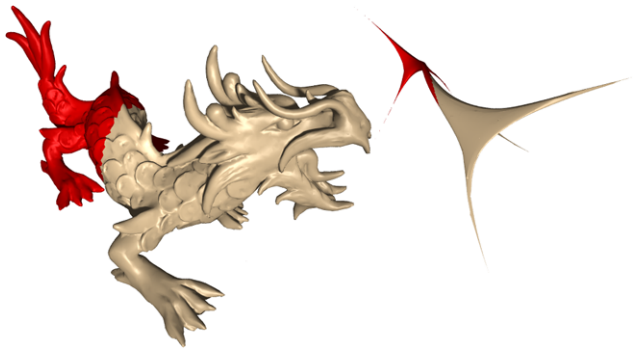


Fig. 5 The dragon model with its eigenskeleton created using 8 eigenvectors

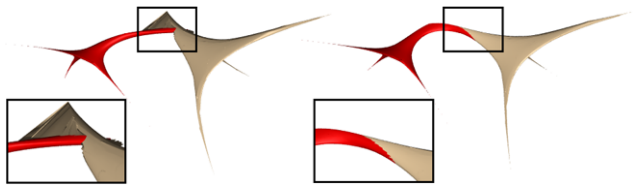


Fig. 6 *Left picture*: A coarse discontinuous initial guess. Rotating the entire region of interest (colored *red*) causes the discontinuity at its boundary. We use the mesh connectivity information from the original mesh to further emphasize this point. Right picture: After Step 2, we obtain a smooth transition across the boundary

(along the boundary of the region of influence R, there is a dramatic, noncontinuous change in the deformation). However, we will see later that in Step 2, our algorithm takes this initial target configuration and produces a much better, smoothly bent eigenskeleton. See Figs. 5 and 6 for an example: in order to bend the body of the dragon, we specify a rotation on the back half of the dragon. We then apply this rotation on the entire region of interest in the eigenskeleton to obtain a target configuration $\tilde{\mathsf{K}}$, as shown in the left image in Fig. 6. Note that Step 2 will produce a nice, global deformation for the eigenskeleton, as shown on the right in Fig. 6.

Observe that the translation-type and rotation-type motions are only high-level guidance for producing the final deformation in Step 2. The final deformation is of course not necessarily rigid. A stretching effect, for example, can be achieved by a simple translation-type motion. From the user's point of view, the amount of work to specify the deformation is very little and rather intuitive, while the algorithm reconstructs a more complex deformation from the user's coarse input.

3.2 Step 2: Eigenskeleton deformation

After Step 1, we have a guess-target configuration $\tilde{\mathsf{K}}$ for the eigenskeleton K. In this step, we wish to compute an improved target deformed eigenskeleton K^* from the guess-target configuration $\tilde{\mathsf{K}}$. In Step 3 described in next section, we will add details back to K^* to obtain a deformed surface M^* for the input surface M. The following diagram illustrates successive structures.

$$\mathsf{M} \xrightarrow[\text{skeleton}]{\text{Get}} \mathsf{K} \xrightarrow[\text{guess target}]{\text{Step 1}} \tilde{\mathsf{K}} \xrightarrow[\text{improve target}]{\text{Step 2}} \mathsf{K}^* \xrightarrow[\text{add details}]{\text{Step 3}} \mathsf{M}^*$$

$$v_i \longrightarrow p_i \longrightarrow \tilde{p}_i \longrightarrow p_i^* \longrightarrow v_i^*$$

Recall that $\tilde{p}_i$ is the position of the ith vertex v_i in the guess-target skeleton $\tilde{\mathsf{K}}$. Now consider the coordinate functions $(\tilde{f}_x, \tilde{f}_y, \tilde{f}_z)$ of the guess-target skeleton $\tilde{\mathsf{K}}$. Note, each function $\tilde{f}_a$, where $a \in \{x, y, z\}$, is a function $\mathsf{M} \to \mathbb{R}$ on the input surface M. The guess configuration $\tilde{\mathsf{K}}$ is often far from being satisfactory. In particular, by cutting the region of influence and simply translating and rotating this part, a discontinuity exists at the boundary of region of influence. In other words, there is no smooth transition across the cut. See the enlarged picture in Fig. 6 left image. This means that the coordinate functions $\tilde{f}_a$ are not smooth across the cut. To get a smooth deformed skeleton, we wish to find a smooth approximation f_a^* for each $\tilde{f}_a$. This will give rise to an improved deformed skeleton K^* with the ith vertex $p_i^* = (f_x^*[i], f_y^*[i], f_z^*[i])$.

To this end, note that since the eigenfunctions ϕ_js of M form a basis for the family of square-integrable functions on M, each $\tilde{f}_a$ can be written as a linear combination of all eigenfunctions ϕ_is for $i = 1, \ldots, n$. Furthermore, eigenfunctions with low eigenvalues are analogous to modes with low-frequency while those with high eigenvalues correspond to high-frequency modes. Since we aim to obtain a smooth reconstruction of $\tilde{f}_a$, we want to ignore high frequency modes. Hence, we find a smooth reconstruction of $\tilde{f}_a$ using only the top m low-frequency eigenfunctions $\phi_1, \ldots, \phi_m$ of M. This is achieved as follows: Suppose $f_a^* = \sum_{j=1}^m \tilde{\alpha}_a^j \phi_j$, and $A_j = (\tilde{\alpha}_x^j, \tilde{\alpha}_y^j, \tilde{\alpha}_z^j)$ for $j = 1, \ldots, m$. We want to find weights $(\tilde{\alpha}_x, \tilde{\alpha}_y, \tilde{\alpha}_z)$ that minimize the following energy function where $\phi_j[i]$ is the value of the jth eigenfunction ϕ_j on the vertex v_i:

$$E = \sum_{i=1}^{n} \left\| \sum_{j=1}^{m} A_j \phi_j[i] - \tilde{p}_i \right\|^2 \tag{1}$$

Intuitively, the discontinuity in the coordinates of guess-target configuration $\tilde{\mathsf{K}}$ requires high frequency eigenfunctions to reconstruct it, and using only low-frequency modes produces smoother f_a^*s, which induces better deformed skeleton K^*. See the right picture of Fig. 6—the skeleton reconstructed from new coordinate weights $(\tilde{\alpha}_x, \tilde{\alpha}_y, \tilde{\alpha}_z)$ after Step 2 shows a smooth transition from the region of interest to the rest.

3.2.1 An alternative interpretation

Before we describe how we minimize the above energy function, we provide an alternative interpretation for the formulation of our energy function. Recall after Step 1, we have a guess-target configuration $\tilde{\mathsf{K}}$ for the eigenskeleton K, and $\tilde{p}_i$ is the position of vertex v_i in this skeleton $\tilde{\mathsf{K}}$. Intuitively, if $\tilde{\mathsf{K}}$ turns out to be the eigenskeleton of an isometric deformation $\tilde{\mathsf{M}}$ of M, then there exist new coordinate weights $(\tilde{\alpha}_x, \tilde{\alpha}_y, \tilde{\alpha}_z)$, such that

$$\left\| \sum_{j=1}^{m} A_j \phi_j[i] - \tilde{p}_i \right\|^2 = 0 \quad \forall v_i \in V$$

where $A_j = (\tilde{\alpha}_x^j, \tilde{\alpha}_y^j, \tilde{\alpha}_z^j)$ represents the jth entry of each $\tilde{\alpha}_a$. This is true because the manifold $\tilde{\mathsf{M}}$ has the same eigenfunctions as M, and its corresponding coordinate functions can be written as a linear combination of the eigenfunctions of $\tilde{\mathsf{M}}$ (i.e., $\phi_1, \ldots, \phi_n$). The new coordinate weights $\tilde{\alpha}_a$ are simply the first m coefficients for $\phi_1, \ldots, \phi_m$ in this linear combination.

If the deformation is not isometric, then we can try to find the best fit $(\tilde{\alpha}_x^j, \tilde{\alpha}_y^j, \tilde{\alpha}_z^j)$ for $j \in [1, m]$ by minimizing the above quantity over all vertices in V, that is, minimizing the energy function as defined in Eq. (1). Experimental results show that our method tends to preserve isometry in practice when such a deformation is possible; see for example Fig. 10 and Table 2. At the same time, since we do not try to preserve the Laplace operator, we can handle nonisometric deformations like stretching in a more natural manner, compared to [25, 26, 29]; see, for example, Fig. 4.

3.2.2 Minimizing the Energy function E

There are $3m$ variables in the energy function in Eq. (1). To minimize E, we compute its gradient with respect to A_k

$$\frac{\partial E}{\partial A_k} = 2 \sum_{i=1}^{n} \phi_k[i] \left(\sum_{j=1}^{m} A_j \phi_j[i] - \tilde{p}_i \right)$$
$$= 2 \left(\sum_{j=1}^{m} A_j \langle \phi_k \cdot \phi_j \rangle - \left(\langle \phi_k \cdot \tilde{f}_x \rangle, \langle \phi_k \cdot \tilde{f}_y \rangle, \langle \phi_k \cdot \tilde{f}_z \rangle \right) \right)$$

where $(\tilde{f}_x, \tilde{f}_y, \tilde{f}_z)$ are the coordinate functions of the guess-target skeleton. Now, setting the partial derivatives to zero for all A_k, we get

$$\sum_{j=1}^{m} \langle \phi_k \cdot \phi_j \rangle A_j = \left(\langle \phi_k \cdot \tilde{f}_x \rangle, \langle \phi_k \cdot \tilde{f}_y \rangle, \langle \phi_k \cdot \tilde{f}_z \rangle \right)$$

which leads to a linear system of equations in the following form: $\Phi A^* = b$, where Φ is an m by m matrix with $\Phi_{i,j} = \langle \phi_i \cdot \phi_j \rangle$,[2] A^* is an m by 3 matrix with $A^*_{i,\cdot} = A_i$ and b is also an m by 3 matrix with the ith row as $(\langle \phi_i \cdot \tilde{f}_x \rangle, \langle \phi_i \cdot \tilde{f}_y \rangle, \langle \phi_i \cdot \tilde{f}_z \rangle)$. Using A^* as coordinate weights, we reconstruct the new deformed eigenskeleton K^*.

3.3 Step 3: Shape recovery

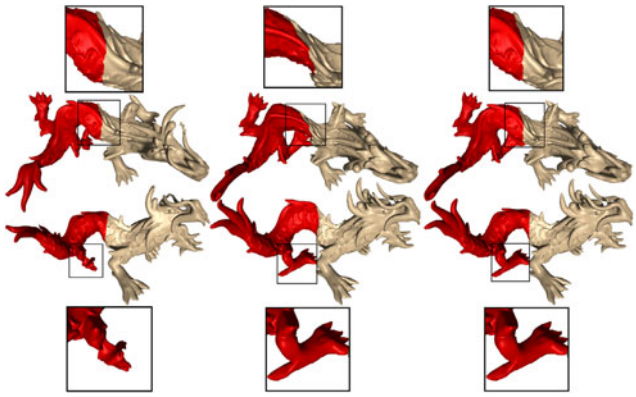
We now have the deformed eigenskeleton K^*. Since we use only the top few eigenvectors for deformation, this skeleton lacks small features and fine details of the original mesh. To obtain the deformed mesh M^*, we need to add appropriate details back to K^*.

In order to keep track of all the shape details, when creating the original eigenskeleton K, we also keep track of the difference between v_i and its reconstruction p_i. We call it the *detail vector* which is given by $d_{v_i} = v_i - p_i$. However, since the mesh is deforming, we cannot simply add d_{v_i} back to $\tilde{p}_i$.

To address this issue, we keep track of d_{v_i} in a *local coordinate frame* around p_i. In particular, for each p_i, we compute three axes that are given by: (i) the normal at p_i, (ii) projection of an edge incident at p_i onto a tangent plane at p_i, and (iii) a third vector orthogonal to the previous two. This frame remains consistent with the local orientation of the vertex. For each detail vector d_{v_i}, we record its coordinates in this local frame, which is the projection of d_{v_i} onto the three axes. After the eigenskeleton is deformed to a new configuration K^*, we compute the new frames, and reconstruct $\tilde{d}_{v_i}$ using the same coordinates but in the new frame. We then obtain the deformed location $\tilde{v}_i$ for vertex v_i by adding the new detail vector back to the skeleton point $\tilde{p}_i$; that is, $\tilde{v}_i = \tilde{p}_i + \tilde{d}_{v_i}$.

A drawback of this local-frame based scheme is that the resulting eigenskeleton becomes very thin near the extremities, with a lot of small features collapsing together, when very few eigenvectors are used. This leads to poor normal estimation around sharp features. See the dragon foot in Fig. 7(a). To overcome this difficulty, we compute two sets of new detail vector $\tilde{d}_{v_i}$: one is obtained by using the local-frames as described above, denoted by $\tilde{d}^{(1)}_{v_i}$; and the other, denoted by $\tilde{d}^{(2)}_{v_i}$, is obtained by simply applying the target deformation transformation computed in Step 1 to the original d_{v_i}. The advantage of $\tilde{d}^{(2)}_{v_i}$ is that it tends to preserve local details. However, just using $\tilde{d}^{(2)}_{v_i}$ alone has the problem that the changes around boundary of R are often dramatic. See the body of the dragon in Fig. 7(b).

[2] In the ideal case, the Laplace–Beltrami operator is symmetric, which makes its eigenvectors orthonormal, and Φ the identity matrix. However, we use area weights when building the Laplace–Beltrami operator, which makes it asymmetric, and Φ a matrix with non-zero off diagonal entries.



(a) Local frame vectors (b) Transformed vectors (c) Interpolated vectors

Fig. 7 Adding details back to the dragon

To get the best out of both strategies, we obtain a final detail vector $\tilde{d}_{v_i}$ by interpolating between the two detail vectors $\tilde{d}_{v_i}^{(1)}$ and $\tilde{d}_{v_i}^{(2)}$. In particular, we assign a large weight to the local-frame based detail vector (i.e., $\tilde{d}_{v_i}^{(1)}$) near the boundary of R and diminish away from the boundary both inside and outside R. We do so because we observed that the normal estimation (and hence $\tilde{d}_{v_i}^{(1)}$) is reliable away from the extremities and near the boundary of the region of interest and that $\tilde{d}_{v_i}^{(2)}$ provides accurate recovery of the sharp features. This works for most models commonly used in the real world, although it is theoretically possible for a mesh and its corresponding skeleton to be very thin even in regions away from the extremities. The details of this interpolation scheme are described in Sect. 4.1. Figure 7 shows results for recovering the details of the deformed dragon using each individual strategies (a, b) and using the integrated strategy (c).

4 Implementation

4.1 Recovery details

For interpolating the detail vectors, we need to assign a weight to each vertex which should depend on how far it is from the boundary of the region of interest. To do this, we need to (1) identify the boundary ∂R of the region of interest R; (2) compute a per-vertex function denoting the distance from the boundary ∂R; and (3) use this function to assign the interpolation weights.

The boundary vertices are identified by considering all vertices in R and simply choosing the ones whose one-ring neighborhood contains vertices that are not in R. We precompute the one-ring neighborhoods on the original mesh just once to reduce computation time during actual deformation of the mesh.

Next, we first compute the following function for each vertex v_i: $g(v_i) = \min_{v \in \partial R} d_g(v_i, v)$, where ∂R is the set of boundary vertices of the region of interest R, and $d_g(v_i, v)$ denotes the geodesic distance between two vertices. Again, we precompute the all-pair geodesic distance matrix once for the original mesh and use it subsequently for all deformations.

Once we have g, we find the approximate diameter of R as $\text{diam}R = \max_{v \in R} g(v)$. We use the diameter to compute two cutoff values $\delta_1 = \frac{\text{diam}R}{8}$, and $\delta_2 = \frac{\text{diam}R}{4}$ The interpolation weights are then computed as

$$w(v_i) = \begin{cases} 1 & \text{if } g(v_i) < \delta_1 \\ 0 & \text{if } g(v_i) > \delta_2 \\ \frac{\delta_2 - g(v_i)}{\delta_2 - \delta_1} & \text{otherwise} \end{cases}$$

The final detail vector at each vertex v_i is then

$$\tilde{d}_{v_i} = \left(w(v_i) \cdot \frac{\tilde{d}_{v_i}^{(1)}}{\|\tilde{d}_{v_i}^{(1)}\|} + \big(1 - w(v_i)\big) \cdot \frac{\tilde{d}_{v_i}^{(2)}}{\|\tilde{d}_{v_i}^{(2)}\|} \right) \cdot \|\tilde{d}_{v_i}^{(1)}\|$$

Note that the two detail vectors $\tilde{d}_{v_i}^{(1)}$ and $\tilde{d}_{v_i}^{(2)}$ have the same length. The above formula simply interpolates their directions to obtain $\tilde{d}_{v_i}$. Intuitively, the closer a point is to the boundary ∂R of the region of interest, the larger role the local-frame detail vector $\tilde{d}_{v_i}^{(1)}$ plays to guarantee smooth transition. When a point is far from ∂R, the skeleton tends to be much thinner, and in this case we rely more on the transformation-based detail vector $\tilde{d}_{v_i}^{(2)}$ to reconstruct $\tilde{d}_{v_i}$.

4.2 Choice of number of eigenvectors

The eigenvectors capture details at different scales. Consequently, the use of different number of eigenvectors for deformation causes changes at different scales.

In general, the eigenskeleton created with only the top few eigenvectors causes shape changes at a global level. To capture local changes, we need a larger number of eigenvectors. Specifically, if the region of interest R is small, then we need more eigenvectors to build the skeleton so that R is reconstructed reasonably well in this skeleton and the change of the corresponding coordinate weights are sufficient to deform R. See Fig. 8 where if we choose too few eigenvectors, the eigenskeleton of the ear collapses into roughly a point, and cannot represent the ear at all. Since deformation is computed for the eigenskeleton, the deformation of the ear cannot be described by such a skeleton. Using more eigenvectors, we can capture the ear in the skeleton and further deform it.

On the other hand, if the region of interest is large, the change usually needs to be spread over a large area. If we now choose too many eigenvectors, minimizing the energy function in Step 2 tries to preserve local details of the

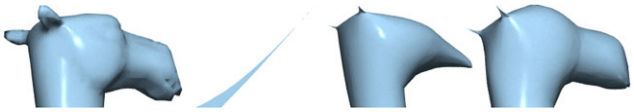


Fig. 8 *Far left*: head of camel. *Right*: Eigenskeleton of the head of the camel constructed using 8, 50, and 300 eigenvectors, respectively

eigenskeleton (as there are more terms, i.e., A_js with large j, describing them). Roughly speaking, the optimization of the weights of the lower eigenvectors is overwhelmed by the large number of higher eigenvectors. Hence, the deformation of the eigenskeleton returned in Step 2 tends to have some dramatic changes for a few points while trying to preserve local details elsewhere. Therefore, in the case of a large region of interest, we need to choose a small number of eigenvectors to build the eigenskeleton so that the weight for global deformation is emphasized.

In summary, the number of eigenvectors n_{ev} to be used to reconstruct the eigenskeleton should be chosen based on the size of R, the region of interest. At the same time, it turns out that the deformation returned by our algorithm is rather robust with respect to n_{ev}, as long as n_{ev} is within a reasonable range. We thus use the following simple strategy to decide n_{ev}. First, compute $\delta_3 = \frac{\text{diam}R}{\text{diam}(V \setminus R)}$, where $\text{diam}(V \setminus R) = \max_{v \notin R} g(v)$ is the approximate diameter of the complement of the region of interest. Now choose n_{ev} as

$$n_{\text{ev}} = \begin{cases} 8 & \text{if } \delta_3 \geq 0.75 \\ 50 & \text{if } 0.75 > \delta_3 \geq 0.1 \\ 300 & \text{otherwise} \end{cases}$$

This simple strategy works well for all the models we experimented with. However, the user can easily override these defaults to choose their own value for n_{ev}.

4.3 Additional modifications

Finally, we observe that since the eigenskeleton can be rather coarse when n_{ev} is small (8 or 50), the local frame estimation sometimes simply becomes too error-prone on $\mathsf{K}^*_{n_{\text{ev}}}$ to recover a smooth shape through the interpolation strategy.

For this reason, we iteratively improve the quality of the eigenskeleton based on the algorithm introduced in Sect. 3 as follows: Recall that K^*_m denotes the deformed eigenskeleton reconstructed using m eigenvectors. Instead of directly recovering the deformed mesh M^* from K^*_m, we first recover another intermediate eigenskeleton $\tilde{\mathsf{K}}_{n'}$ from $\mathsf{K}^*_{n_{\text{ev}}}$, with $n' > n_{\text{ev}}$ using Algorithm 1. This is achieved by using the detail vectors to record the change from $\tilde{\mathsf{K}}_{n_{\text{ev}}}$ to $\tilde{\mathsf{K}}_{n'}$, instead of $\tilde{\mathsf{K}}_{n_{\text{ev}}}$ to the original mesh M. In particular, in our software, $n_{\text{ev}} = 8$ or 50, and $n' = 300$ (this iterative approach is not needed if $n_{\text{ev}} = 300$). The result is an eigenskeleton that already captures the main deformation, and that also contains sufficient details.

Fig. 9 Adding details back to the dragon. *Left*: Directly from eigenskeleton. *Right*: After iterative improvement

Next, we feed $\tilde{\mathsf{K}}_{n'}$ as the coarse-guess configuration to the linear solver in Step 2 to obtain a new deformed eigenskeleton $\mathsf{K}^*_{n'}$. This is done to smooth out any errors that may have been introduced due to poor local frame estimation on $\mathsf{K}^*_{n_{\text{ev}}}$. We then use this new eigenskeleton $\mathsf{K}^*_{n'}$ and local-frame based detail estimation (instead of the interpolation method) to recover the shape-detail of the deformed mesh M^*. See Fig. 9 for an example. The final deformation algorithm for the case that $n_{\text{ev}} = 8$ or 50 is summarized in the following diagram.

Iteration 1:

$$\mathsf{K}_{n_{\text{ev}}} \xrightarrow{\text{Step 1}} \tilde{\mathsf{K}}_{n_{\text{ev}}} \xrightarrow{\text{Step 2}} \mathsf{K}^*_{n_{\text{ev}}} \xrightarrow[\text{Interpolation based}]{\text{Step 3}} \tilde{\mathsf{K}}_{n'}$$

Iteration 2:

$$\tilde{\mathsf{K}}_{n'} \xrightarrow{\text{Step 2}} \mathsf{K}^*_{n'} \xrightarrow[\text{local-frame based}]{\text{Step 3}} \mathsf{M}^*$$

For the case where $n_{\text{ev}} = 300$, the original Algorithm 1 is applied as before. We remark that potentially one can perform more iterations to improve the deformation quality. However, we observe in practice that two iterations provide a good trade-off between quality and simplicity/efficiency.

4.4 Interactivity

To make the software interactive, we precompute the eigenvectors for the mesh along with the matrix Φ since it depends on the original mesh only. Notice that Φ is symmetric, and hence can be factored using Cholesky decomposition. We also precompute the all-pairs geodesic distance matrix used for interpolating detail vectors. To maintain interactive rates, we only deform the eigenskeleton. Once the user is satisfied with the shape of the eigenskeleton, the details are added. When deforming the eigenskeleton, the right-hand side (b) for our linear-solver can be quickly computed by multiplying the matrix of eigenvectors with a matrix containing the coarse guess. We can then compute the new coordinate weights by performing simple backward and forward substitutions. The entire process is simple and can be

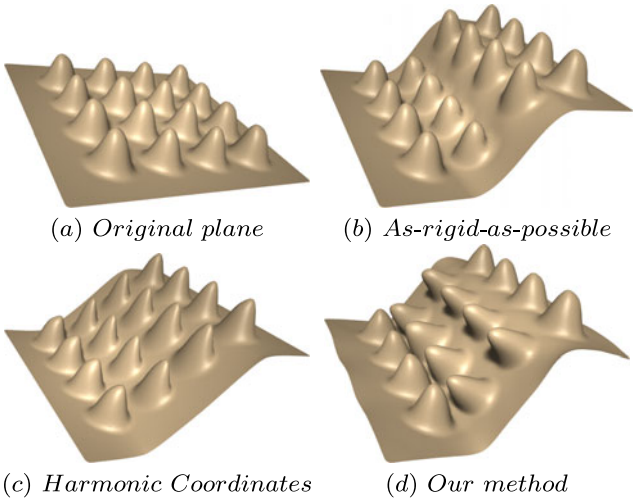


Fig. 10 Bending a bumpy plane (dense mesh)

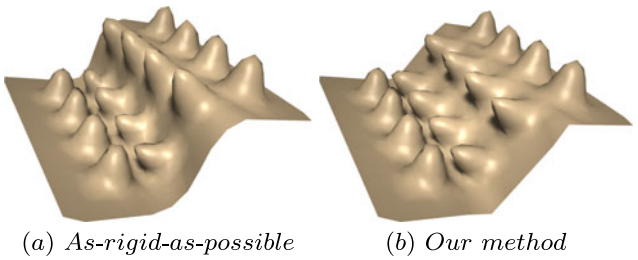


Fig. 11 Bending a bumpy plane (coarse mesh)

computed in real-time. Adding details can be a little slow (see Table 1) since we need to compute the normal for each vertex, and hence is separated from the interactive part.

5 Results

We implemented our deformation algorithm using C, OpenGL and MATLAB. For comparisons, we wrote our own code for as-rigid-as-possible deformations [29] and used the implementation of cage-based deformation using harmonic coordinates provided in open-source software called BLENDER. For spectral surface deformation, we used the code provided by the authors.

Figure 3 compares our method with harmonic coordinates and as-rigid-as-possible deformations. For as-rigid-as-possible deformation, red dots denote the fixed vertices, while yellow dots represent the vertices that are moved. The partial cages used for deforming using harmonic coordinates are depicted using black edges. For our method, the red portions are the regions of interest. Figures 10 and 11 show the results of bending a plane with smooth bumps using different techniques. Harmonic coordinates are not able to orient the details correctly while for as-rigid-as-possible, the quality of the deformation seems to depend on mesh density.

Table 1 Timing data (in seconds) for our algorithm

Model (# vertices)	n_{ev}	Step1 & 2	Step 3
Armadillo (25k)	50	0.018	0.125
Dragon (22.5k)	8	0.013	0.161
Camel (7k)	8	0.002	0.049
	300	0.012	0.013
Plane (10k)	50	0.016	0.061
Bar (13.5k)	50	0.017	0.074
Children (20k)	50	0.017	0.095

Table 2 Comparison of relative RMS errors in deformations using as-rigid-as-possible (ARAP) and our method (ED)

Model	ARAP	ED
Armadillo (Stretch Arm)	0.0023	0.0022
Armadillo (Bent Knee)	0.001	0.0007
Armadillo (Combined)	0.0052	0.0021
Plane	0.0028	0.0022

Fig. 12 Moving the arm of Neptune using our method and spectral mesh deformation (SMD)

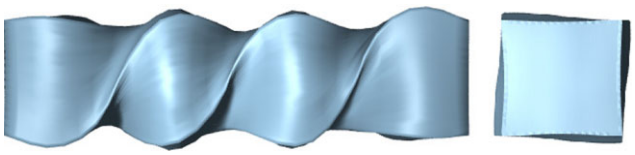


Fig. 13 Twisting a bar using our method

The timing data for different stages of our algorithm are presented in Table 1. Timings of Steps 1 and 2 are coupled together since they are used in each step of interactive deformation. Step 3 is used after the user is satisfied with the shape of the skeleton. Table 2 compares the root mean square error in edge lengths. Our method introduces very little error in edge lengths, similar to as-rigid-as-possible approach which aims to optimize such error. Figure 13 shows the result of twisting a bar using our method, while Fig. 14 shows that we can handle meshes of arbitrary genus. More

Fig. 14 Editing the dancing children

deformations using our method can be found in the video submitted with this paper.

We also present comparisons with spectral mesh deformation in Figs. 4, 12. The results of spectral mesh deformation are global and cannot be constrained to small regions. For example, in Fig. 4, even though only the arm was stretched, the entire mesh got deformed in an attempt to preserve the volume and the Laplace operator of the mesh. Our method is able to handle such deformations more naturally. Also, the detail recovery method used in spectral mesh deformation can introduce artifacts into the deformed mesh. For example, in Fig. 12, although only the arm was moved, the staff of Neptune got slightly deformed as well. Even the hand looks unnatural after the deformation. This happens because 100 eigenvectors are not enough to capture the finer details of the model.

Acknowledgements We would like to thank the anonymous reviewers for their comments, and also the authors of [26] for providing the software implementation of their work. Meshes used in this paper were obtained from AIM@Shape Shape Repository. This work is supported by the National Science Foundation Grants CCF-0830467 and CCF-0747082.

References

1. Baran, I., Popović, J.: Automatic rigging and animation of 3D characters. In: Proc. SIGGRAPH'07, pp. 72:1–72:8 (2007)
2. Belkin, M., Sun, J., Wang, Y.: Discrete Laplace operator on meshed surfaces. In: Proc. SCG'08, pp. 278–287 (2008)
3. Ben-Chen, M., Weber, O., Gotsman, C.: Variational harmonic maps for space deformation. In: Proc. SIGGRAPH'09, pp. 34:1–34:11 (2009)
4. Botsch, M., Kobbelt, L.: Real-time shape editing using radial basis functions. Comput. Graph. Forum **24**(3), 611–621 (2005)
5. Botsch, M., Pauly, M., Gross, M., Kobbelt, L.: Primo: coupled prisms for intuitive surface modeling. In: Proc. SGP'06, pp. 11–20 (2006)
6. Botsch, M., Sorkine, O.: On linear variational surface deformation methods. IEEE Trans. Vis. Comput. Graph. **14**(1), 213–230 (2008)
7. Desbrun, M., Meyer, M., Schröder, P., Barr, A.H.: Implicit fairing of irregular meshes using diffusion and curvature flow. In: Proc. SIGGRAPH'99, pp. 317–324 (1999)
8. Dey, T.K., Ranjan, P., Wang, Y.: Convergence, stability, and discrete approximation of Laplace spectra. In: Proc. SODA'10, pp. 650–663 (2010)
9. Du, H., Qin, H.: Medial axis extraction and shape manipulation of solid objects using parabolic PDEs. In: Proc. ACM Sympos. Solid Modeling Appl.'04, pp. 25–35 (2004)
10. Floater, M.S.: Mean value coordinates. Comput. Aided Des. **20**(1), 19–27 (2003)
11. Floater, M.S., Kos, G., Reimers, M.: Mean value coordinates in 3D. Comput. Aided Des. **22**(7), 623–631 (2005)
12. Hildebrandt, K., Polthier, K.: On approximation of the Laplace–Beltrami operator and the Willmore energy of surfaces. In: Proc. SGP'11, pp. 1513–1520 (2011)
13. Hildebrandt, K., Schulz, C., Tycowicz, C.V., Polthier, K.: Interactive surface modeling using modal analysis. ACM Trans. Graph. **30**(5), 119:1–119:11 (2011)
14. Jacobson, A., Baran, I., Popović, J., Sorkine, O.: Bounded biharmonic weights for real-time deformation. In: Proc. SIGGRAPH'11, pp. 78:1–78:8 (2011)
15. Joshi, P., Meyer, M., DeRose, T., Green, B., Sanocki, T.: Harmonic coordinates for character articulation. In: Proc. SIGGRAPH'07, pp. 71:1–71:10 (2007)
16. Ju, T., Schaefer, S., Warren, J., Desbrun, M.: A geometric construction of coordinates for convex polyhedra using polar duals. In: Proc. SGP'05, pp. 181–186 (2005)
17. Karni, Z., Gotsman, C.: Spectral compression of mesh geometry. In: Proc. SIGGRAPH'00, pp. 279–286 (2000)
18. Kobayashi, K.G., Ootsubo, K.: T-ffd:free-form deformation by using triangular mesh. In: Proc. Sympos. Solid Modeling Appl.'03, pp. 226–234 (2003)
19. Langer, T., Belyaev, A., Seidel, H.-P.: Spherical barycentric coordinates. In: Proc. SGP'06, pp. 81–88 (2006)
20. Levy, B.: Laplace–Beltrami eigenfunctions: towards an algorithm that understands geometry. In: IEEE Internat. Conf. on Shape Modeling Appl.'06 (2006). Invited talk
21. Lipman, Y., Levin, D., Cohen-Or, D.: Green coordinates. In: Proc. SIGGRAPH'08, pp. 78:1–78:10 (2008)
22. MacCracken, R., Joy, K.I.: Free-form deformations with lattices of arbitrary topology. In: Proc. SIGGRAPH'96, pp. 181–188 (1996)
23. Pinkall, U., Polthier, K.: Computing discrete minimal surfaces and their conjugates. Exp. Math. **2**(1), 15–36 (1993)
24. Reuter, M., Wolter, F.-E., Peinecke, N.: Laplace–Beltrami spectra as "shape-dna" of surfaces and solids. Comput. Aided Des. **38**(4), 342–366 (2006)
25. Rong, G., Cao, Y., Guo, X.: Spectral surface deformation with dual mesh. In: Proc. Internat. Conf. on Comput. Animation and Social Agents'08, pp. 17–24 (2008)
26. Rong, G., Cao, Y., Guo, X.: Spectral mesh deformation. Vis. Comput. **24**(7–9), 787–796 (2008)
27. Sederberg, T.W., Parry, S.R.: Free-form deformation of solid geometric models. In: SIGGRAPH'86, pp. 151–160 (1986)
28. Sorkine, O.: Differential representations for mesh processing. Comput. Graph. Forum **25**(4), 789–807 (2006)
29. Sorkine, O., Alexa, M.: As-rigid-as-possible surface modeling. In: Proc. SGP'07, pp. 109–116 (2007)
30. Sorkine, O., Lipman, Y., Cohen-Or, D., Alexa, M., Rössl, C., Seidel, H.-P.: Laplacian surface editing. In: Proc. SGP'04, pp. 179–188 (2004)
31. Warren, J.: Barycentric coordinates for convex polytopes. Adv. Comput. Math. **6**(2), 97–108 (1996)
32. Weber, O., Sorkine, O., Lipman, Y., Gotsman, C.: Context-aware skeletal shape deformation. Comput. Graph. Forum 265–274
33. Yoshizawa, S., Belyaev, A.G., Seidel, H.-P.: Free-form skeleton-driven mesh deformations. In: Proc. ACM Sympos. Solid Modeling Appl.'03, pp. 247–253 (2003)
34. Zhang, H., van Kaick, O., Dyer, R.: Spectral mesh processing. Comput. Graph. Forum **29**(6), 1865–1894 (2010)
35. Zhou, K., Huang, J., Snyder, J., Liu, X., Bao, H., Guo, B., Shum, H.-Y.: Large mesh deformation using the volumetric graph Laplacian. ACM Trans. Graph. **24**(3), 496–503 (2005)

Tamal K. Dey is a professor of computer science at The Ohio State University. His research interests include computational geometry, computational topology, and their applications in graphics and geometric modeling. He finished his Ph.D. from Purdue University in 1991 and has held academic positions at various institutions. He authored a book "Curve and surface reconstruction: Algorithms with mathematical analysis" published by Cambridge University Press and led the development of well- known software called Cocone and DelPSC for surface reconstruction and mesh generation. Details can be found at http://www.cse.ohio-state.edu/~tamaldey.