EUROGRAPHICS 2014 / B. Lévy and J. Kautz
(Guest Editors)

Volume 33 (2014), Number 2

Accurate and Efficient Lighting for Skinned Models

Marco Tarini[1,2] Daniele Panozzo[3] Olga Sorkine-Hornung[3]

[1] Università dell'Insubria, Varese [2] ISTI-CNR Pisa [3] ETH Zurich

Abstract

In the context of real-time, GPU-based rendering of animated skinned meshes, we propose a new algorithm to compute surface normals with minimal overhead both in terms of the memory footprint and the required per-vertex operations. By accounting for the variation of the skinning weights over the surface, we achieve a higher visual quality compared to the standard approximation ubiquitously used in video-game engines and other real-time applications. Our method supports Linear Blend Skinning and Dual Quaternion Skinning. We demonstrate the advantages of our technique on a variety of datasets and provide a complete open-source implementation, including GLSL shaders.

1 Introduction

Skinning is ubiquitously used in movies and games to animate non-rigid bodies such as virtual characters, including their clothes and other deformable parts [KSO10]. It decouples the character being animated from the animation itself by employing a low-dimensional set of degrees of freedom to drive the animation of a potentially complex, detailed shape. In video games, the de-facto standard skinning methods are *Linear Blend Skinning* (LBS) [MTLT88] and *Dual Quaternion Skinning* (DQS) [KCZO07]. Despite the existing variety of more elaborate schemes with superior expressiveness and shape quality (e.g., [HYC*05, FO06, YSZ06, JS11, JBK*12, VBG*13]), many of which also explicitly take normal deformation into account [KJP02, MG03, JT05, DSP06, RJ07, BJ07, WPP07], modern 3D games use LBS or DQS in practice. This is because these two methods fit very well in the standard GPU rendering pipeline, and they deliver acceptable animation quality at a small overhead cost, both in terms of computation (at rendering time) and memory usage (main storage and footprint during rendering).

Surprisingly, despite the vast popularity of LBS and DQS as shape deformers, the local illumination of the animated models is routinely computed by using a rather crude approximation of the correct normals and tangent frames. This leads to visible shading discrepancies and may negatively impact the perception of the animated shape and its motion (see Fig. 1). The approximation error stems from ignoring the spatially-varying, non-rigid deformation component of skinning when updating the normals and tangents during the animation, as traditionally done in order to avoid expensive computations.

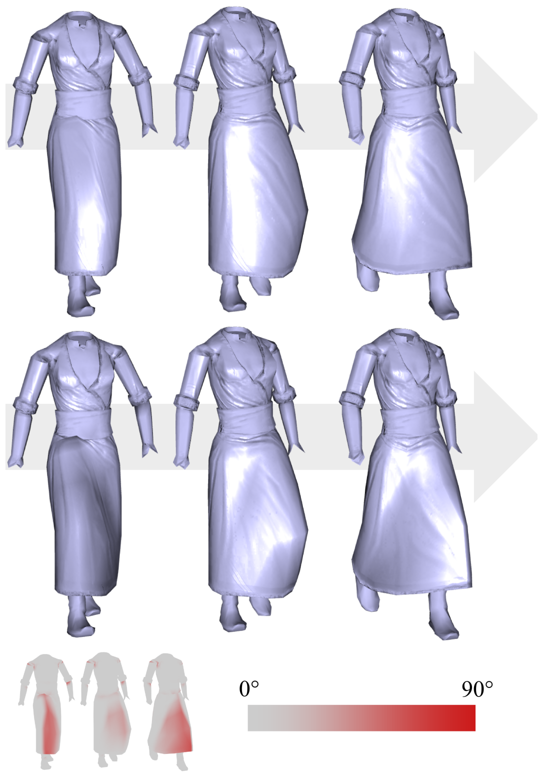

Figure 1: *Lighting a skinned low-poly, normal-mapped model. Top sequence: the standard method produces inaccurate normals that makes the skirt look flat. Mid sequence: our algorithm produces an accurate lighting which better conveys the shape of the skirt. Bottom: differences in normals. See also attached video.*

In this paper, we show how to accurately compute normals for LBS and DQS without compromising the simplicity and GPU-based efficiency of the original method, both in terms of computation times and bandwidth.

This paper makes the following contributions:

1. We show that the standard approximation of normals and tangents currently used in all major game engines produces visibly inaccurate results;
2. We propose a new algorithm to update normals for LBS and DQS skinning, with higher quality and minimal overhead compared to the standard algorithm;
3. We offer a simple and efficient implementation of our algorithm in form of a ready-to-use GLSL shader that can be easily integrated into existing game engines. The open-source interactive demo is included in the accompanying material.

2 Preliminaries

We briefly recap the basics of LBS and DQS and introduce our notation along the way.

Models animated using skinning consist of three parts: the skeleton, the surface (or skin) mesh, and the binding of the mesh to the skeleton via a set of scalar weight functions.

The *skeleton* $\mathcal{S}$ is a low-dimensional set of degrees of freedom that control the animation. It can be thought of as a collection of abstract *bones*, where each bone $s \in \mathcal{S}$ is associated with an affine transformation $\mathbf{M}^t[s]$ in each frame t of the animation. $\mathbf{M}^t[s]$ is typically a rotation plus a translation ($\mathbf{M}^t[s](\mathbf{x}) = \mathbf{R}^t[s]\,\mathbf{x} + \mathbf{t}^t[s]$). In the rest pose, the bones induce no transformations, so that $\mathbf{M}^0[s]$ is the identity transformation. Throughout the paper, we denote quantities associated with the rest state configuration by a 0-superscript and the current frame by t-superscript. Notably, in LBS the transformations $\mathbf{M}[s]$ are represented by 4×4 affine matrices, and in DQS by unit-norm dual quaternions.

We denote by $\mathcal{R}$ the *rigged mesh* animated through the skeleton; we assume $\mathcal{R}$ is a triangle mesh with combinatorial vertex set $\{1, 2, \dots, n\}$. We denote the *position* of vertex i in the rest pose by $\mathbf{p}_i^0$, and the position in the current animation frame by $\mathbf{p}_i^t$ ($\mathbf{p}_i^0, \mathbf{p}_i^t \in \mathbb{R}^3$). The mesh $\mathcal{R}$ is rigged, or bound, to the skeleton $\mathcal{S}$ via a set of scalar weight functions $\omega[s]$ defined on the mesh vertices, such that for each bone $s \in \mathcal{S}$, the weight $\omega_i[s]$ indicates the amount of influence the bone s has on vertex i.

2.1 Deforming skinned models

A *pose t* of a skinned model (or rather, of its skeleton) is a given set of bone transformations $\{\mathbf{M}^t[s],\ s \in \mathcal{S}\}$. Each animation frame is generated by inducing these transformations onto the skin mesh using the weights. In the LBS formulation, each mesh vertex i gets a new position $\mathbf{p}_i^t$ in the current frame as

$$\mathbf{p}_i^t = F_i^t(\mathbf{p}_i^0) = \left(\sum_{s \in \mathcal{B}_i} \omega_i[s]\, \mathbf{M}^t[s] \right) \mathbf{p}_i^0 = \mathbf{T}_i^t\, \mathbf{p}_i^0. \qquad (1)$$

Here, $\mathcal{B}_i = \{s \in \mathcal{S},\ \omega_i[s] \neq 0\}$ is the set of bones with non-zero weights influencing vertex i. To ensure affine invariance of the skinning deformations, the weight functions partition unity: $\forall i \in \mathcal{R},\ \sum_{s \in \mathcal{S}} \omega_i[s] = 1$. In the DQS formulation, the linear combination of transformations in (1) is replaced with dual quaternion blending (followed by re-normalization); this way of combining rotations and translations has the advantage of better local volume preservation.

The above formula needs to be computed for each mesh vertex, and at 30 fps or higher in modern games. The performance is achieved by parallel computation on the GPU and optimization of the bandwidth, i.e., minimization of the amount of data that needs to be sent to the GPU for each frame. All rest-shape vertex positions $\mathbf{p}^0$ and the scalar per-vertex weights $\omega_i[s]$ are stored on the GPU memory ahead of time. In real time, for each animation frame the bone transformations $\mathbf{M}^t[s]$ are sent to the GPU, typically requiring 12 floats times the number of bones. Then, the new vertex coordinates are computed on the GPU in parallel using (1) or its DQS version, in the vertex shader. This is extremely efficient and requires only a small amount of information to be sent to the GPU per frame, independent of the mesh complexity.

Since the vertex shader has limited capacity, the summation in (1) can only contain a small number of terms $N_{\max}$ (typically $N_{\max} = 4$). Hence the weight functions need to be sparse, such that $|\mathcal{B}_i| \leqslant N_{\max}$. If a skinned model does not have this property, weight sparsification can be applied beforehand [LS10].

2.2 Lighting of skinned models

The rest-shape of the skinned model $\mathcal{R}$ is supplied with attributes to facilitate its rendering, such as per-vertex (u, v)-coordinates into a set of textures, including normal maps. Each vertex i has an associated unit normal $\mathbf{n}_i^0$ and two unit tangent vectors $\mathbf{t}_i^0$ and $\mathbf{b}_i^0$. The latter two vectors are needed for tangent-space normal-mapping (a popular technique in 3D video-games) and sometimes for lighting equations with non-isotropic BRDFs; they are typically computed as the derivatives of the (u, v) mapping. Note that $\mathbf{n}_i^0$ is needed even in the presence of a normal map, since its normals are expressed in tangent space.

To properly render the deformed mesh in any animation frame t, we need the deformed normals $\mathbf{n}_i^t$ and the tangents $\mathbf{t}_i^t, \mathbf{b}_i^t$ as well. Computing them from the geometry of the deformed mesh is not practical on the GPU or even desirable, as the "normals" of the relatively low-poly models used in games often do not coincide with the geometric normals of the surface. Instead, the normals of the current animation frame must be recovered by “warping” the normals supplied with the rest-shape.

Given the deformation mapping F_i^t (see Eq. (1)), the tangent spaces on the surface are transformed by the Jacobian of F_i^t, denoted by $\mathbf{J}_i^t$, and the normals are deformed by ${\mathbf{J}_i^t}^{-T}$:

$$\mathbf{n}_i^t = {\mathbf{J}_i^t}^{-T}\,\mathbf{n}_i^0. \tag{2}$$

The deformation is commonly considered locally rigid, and thus $\mathbf{J}_i^t$ is approximated with $\mathbf{T}_i^t$, as Eq. (1) suggests. Since $\mathbf{T}_i^t$ is close to orthonormal, it also approximates ${\mathbf{J}_i^t}^{-T}$, yielding:

$$\mathbf{n}_i^t \approx \mathbf{T}_i^t\,\mathbf{n}_i^0. \tag{3}$$

(with a slight abuse of notation: whenever a homogeneous matrix multiplies a 3-vector, we mean that only its 3×3 major is used).

Similarly, in DQS, $\mathbf{n}_i^t$ is simply computed by the rotation described by the primal part of the blended dual quaternion.

This approximation is efficient, but it is only accurate if the bone weights are constant around vertex i. Fig. 2 illustrates in 2D an intuitive counterexample. In practice, the errors in the approximated normals can lead to perceivably incorrect lighting in the parts of the model that deform non-rigidly during the animation, as shown e.g. in Fig. 1, 5, 6, 7, 8 and in the accompanying videos.

The mathematically exact computation of ${\mathbf{J}_i^t}^{-T}$ is possible. A formulation to do so, limited to LBS, is reported in [MMG06], and we derive an equivalent in Sec. 3. However, as we detail in the following section, this direct approach suffers from several practical problems, such as computational and memory overhead, as well as frequent numerical instability. We conjecture that these drawbacks are the reason why Eq. (3) is still universally adopted and, despite its inaccuracy, is the current *de facto* standard in video games. In this paper we propose a solution to accurately recover the values of $\mathbf{n}_i^t$ which successfully addresses all these problems: computational and, more importantly, memory overheads are halved (or less, depending on $N_{\max}$), it is numerically robust, artifact free, and, if required, it is easily extended to DQS.

3 Exact skinning Jacobian

In this section we derive the Jacobian of the skinning deformation for the LBS case. The extension to DQS is discussed in the next section.

3.1 Per-triangle LBS Jacobian

Let us consider a single triangle $\mathcal{T}$ of $\mathcal{R}$ in rest pose, formed by mesh vertices i, j, k; denote by $\hat{\mathbf{n}}$ the geometric normal of $\mathcal{T}$. The three corners of $\mathcal{T}$ are $\mathbf{p}_i^0$, $\mathbf{p}_j^0$ and $\mathbf{p}_k^0$, and are associated to weights $\omega_i[s], \omega_j[s], \omega_k[s]$ for bone $s \in \mathcal{S}$.

For our purposes, we can consider the weights to be linearly interpolated functions inside $\mathcal{T}$. Note that this stretches

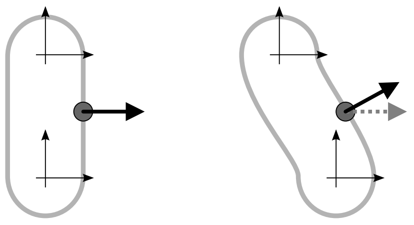

Figure 2: *Conceptual sketch showing the problem with the conventional way to transform normals with LBS or DQS. A simple “pill shaped” mesh is rigged on a skeleton with two bones. Left: rest pose. Right: a pose where the lower bone is just translated to the right, keeping its orientation. In the conventional way (dotted arrow), the normal for the shown point would be transformed by an interpolation of the two rotations, which in this case are both identities. Only when the effect of the variations of the weights over the surface is kept in account, then the correct transformed normals can be recovered (full arrow).*

the semantic of skinning a little: if each point on $\mathcal{T}$ was to be transformed by Eq. (1) with its interpolated weights, then the resulting shape would be in general curved. Instead, in GPU-based skinning, only the vertices are transformed and then linearly connected, producing a deformed, *flat* triangle (weights are never actually interpolated). However, the hypothetical curved triangle and the actual flat triangle are similar in shape.

The interpolated weights for point $\mathbf{p}$, an arbitrary point on triangle $\mathcal{T}$, can be expressed as a linear combination:

$$\omega[s]\,(\mathbf{p}) = \hat{\mathbf{a}}[s]^T\,\mathbf{p}, \tag{4}$$

where $\mathbf{p} \in \mathbb{R}^4$ is represented in homogeneous coordinates, and $\hat{\mathbf{a}}[s] \in \mathbb{R}^4$ is a constant vector associated with $\mathcal{T}$ which solves the following linear system:

$$\begin{cases} \hat{\mathbf{a}}[s]^T\,\mathbf{p}_i^0[s] &= \omega_i[s] \\ \hat{\mathbf{a}}[s]^T\,\mathbf{p}_j^0[s] &= \omega_j[s] \\ \hat{\mathbf{a}}[s]^T\,\mathbf{p}_k^0[s] &= \omega_k[s] \\ \hat{\mathbf{a}}[s]^T\,\hat{\mathbf{n}} &= 0. \end{cases} \tag{5}$$

The first three equations prescribe the weights at $\mathcal{T}$'s vertices to be the assigned ones; the forth equation states that displacing $\mathbf{p}$ along the normal of $\mathcal{T}$, the associated weight remains constant. Since the weight function $\omega[s]$ is linear inside $\mathcal{T}$, its gradient there is constant and equals $\hat{\mathbf{a}}[s]$.

The skinning deformation equation for $\mathbf{p}$, is given by plugging Eq. (4) in Eq. (1):

$$F^t(\mathbf{p}) = \sum_{s \in S} \mathbf{M}^t[s]\,\mathbf{p}\,(\hat{\mathbf{a}}[s]^T\,\mathbf{p}) \tag{6}$$

The Jacobian $\mathbf{J}^t = \partial F^t / \partial \mathbf{p}$ can then be written as:

$$\begin{aligned}\mathbf{J}^t &= \sum_{s\in\mathcal{S}} \left(\mathbf{M}^t[s]\,(\hat{\mathbf{a}}[s]^T\mathbf{p}) + \mathbf{M}^t[s]\,\mathbf{p}\,\hat{\mathbf{a}}[s]^T\right) = \\ &= \mathbf{T}^t + \sum_{s\in\mathcal{S}} \left(\mathbf{q}^t[s]\,\hat{\mathbf{a}}[s]^T\right),\end{aligned} \tag{7}$$

where $\mathbf{q}^t[s] = \mathbf{M}^t[s]\,\mathbf{p}$. Eq. (7) can be intuitively understood as follows. The local deformation of the surface during skinning is comprised of two parts: the transformation defined by the blending of bone transformations, which is rigid or almost rigid, and a local deformation due to the spatial variation of the bone weights, which is non-rigid. The first term of Eq. (7), i.e. the one used in the conventional approximation in Eq. (3), only captures the former effect.

3.2 Per-vertex LBS Jacobian

In Eq. (4), the vectors $\hat{\mathbf{a}}[s]$ are defined per mesh triangle, whereas in realtime rendering pipelines mesh attributes are typically defined per vertex, to take advantage of GPU mechanisms like Vertex Arrays. It is also more memory-efficient, since a mesh typically has twice as fewer vertices as triangles.

Following standard practice in discrete differential geometry [CdGDS13], we define the Jacobian $\mathbf{J}_i^t$ on mesh vertex i by integrating the relevant quantities of the incident triangles over the Voronoi cell of i. Essentially, we propose to compute per-vertex $\mathbf{a}_i[s]$ by weighted averaging of the $\hat{\mathbf{a}}[s]$ over the neighboring triangles, using the well-known cotangent weights. The averaging has the advantage of concealing the piecewise-linear nature of the mesh by making the shading smoother (Fig. 3).

A caveat of this approach is that a vertex i may end up having a nonzero-valued $\mathbf{a}_i[s]$ for a bone $s \notin \mathcal{B}_i$. This happens when at least one vertex in the one-ring of i has a nonzero weight for bone s. In this case, the bone index s must be added to $\mathcal{B}_i$, although with a corresponding weight $\omega_i[s] = 0$. This can occasionally result in overstepping the shader limit ($|\mathcal{B}_i| > N_{\max}$). In typical rigged models, this is very rare, because the influence of a bone is localized and varies smoothly. When the problem does occur, we discard the bones $s \notin \mathcal{B}_i$ with the smallest magnitude of $\mathbf{a}_i[s]$, thus accepting a local approximation.

3.3 Discussion

Updating the normals using the Jacobian as described above requires the following in a GPU-based setting: as a preprocess, storing the $\mathbf{a}_i[s]$ for each vertex i on the GPU memory; at rendering time, computing the Jacobian as above and inverting it, for each vertex i. In terms of memory, we thus have to store $3N_{\max}$ additional floats per vertex (i.e., typically 12 floats), and in terms of computation per-frame, we more than double the entire per-vertex workload of skinning. In scenarios like video-games, where the per-vertex computation and

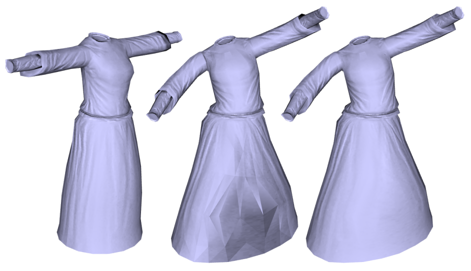

Figure 3: *In the rest pose (left), the piecewise-linear nature of the mesh is concealed by usage of per-vertex normals or tangents. However, if we use per-triangle values of $\hat{\mathbf{a}}[s]$ to compute accurate shading during animation, the triangle structure emerges (middle). Employing the averaged per-vertex $\mathbf{a}_i[s]$ (right) results in smoother illumination.*

memory budget is extremely scarce, these overheads can be very inconvenient. In the case of DQS, the situation would be even worse, since a closed form for $\mathbf{J}^{-T}$ would be far more complex to evaluate. Additionally, our findings show that the straightforward approach above is prone to frequent numerical stability problems (see Sec. 4.3), and, independently of those, it may produce undesirable lighting effects due to the occasional inversion of normals (see Sec. 4.4). In the following section we propose a way to overcome all of these drawbacks.

4 Efficient and robust transformation of normals

We aim to efficiently and robustly transform the normal and tangent vectors in a standard GPU-based skinning pipeline.

4.1 Avoiding inverting the Jacobian

Instead of computing the inverse-transpose of the Jacobian and then multiplying it with the rest-pose normal, a more efficient approach is to transform a pair of tangent vectors $\mathbf{t}_i, \mathbf{b}_i$, and then take their cross product to obtain the transformed normal:

$$\mathbf{t}_i^t = \mathbf{J}_i^t\,\mathbf{t}_i^0, \quad \mathbf{b}_i^t = \mathbf{J}_i^t\,\mathbf{b}_i^0 \;\Rightarrow\; \mathbf{n}_i^t = \mathbf{t}_i^t \times \mathbf{b}_i^t. \tag{8}$$

(The normal should then also be renormalized.)

Since, as mentioned, the transformed tangent directions $\mathbf{t}_i^t$ and $\mathbf{b}_i^t$ are required anyway in any standard rendering pipeline, the only additional computational burden is limited to a cross product.

This approach requires that $\mathbf{t}_i^0$ and $\mathbf{b}_i^0$ are both orthogonal to $\mathbf{n}_i^0$ in the rest shape $\mathcal{R}$. This can be enforced in preprocessing. We cannot override the original per-vertex normals $\mathbf{n}_i^0$ since this would change the intended shading, but we can safely project the per-vertex tangent directions onto the plane

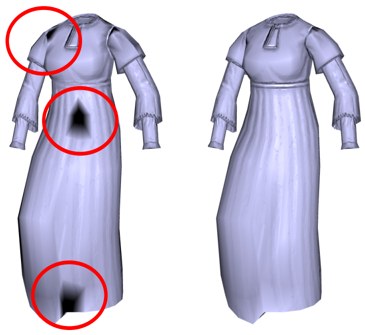

Figure 4: *The cancellation problems of a naive implementation of Eq. (7) result in artifacts (left), which disappear when the countermeasure explained in Sec. 4.3 is enabled (right). See also attached video.*

orthogonal to $\mathbf{n}_i^0$:

$$\begin{aligned}\mathbf{t}_i^0 &\leftarrow \mathbf{n}_i^0 \times \mathbf{t}_i^0 \times \mathbf{n}_i^0 \\ \mathbf{b}_i^0 &\leftarrow \mathbf{n}_i^0 \times \mathbf{b}_i^0 \times \mathbf{n}_i^0 .\end{aligned} \tag{9}$$

4.2 Avoiding the explicit computation of the Jacobian

We can avoid the explicit computation of the matrices $\mathbf{J}_i^t$ by substituting Eq. (7) into Eq. (8):

$$\begin{aligned}\mathbf{t}_i^t &= \mathbf{T}_i^t\,\mathbf{t}_i^0 + \sum_{s\in\mathcal{B}_i} \alpha_i[s]\,\mathbf{q}_i^t[s] \\ \mathbf{b}_i^t &= \mathbf{T}_i^t\,\mathbf{b}_i^0 + \sum_{s\in\mathcal{B}_i} \beta_i[s]\,\mathbf{q}_i^t[s]\end{aligned} \tag{10}$$

where

$$\alpha_i[s] = \mathbf{a}_i[s]\cdot\mathbf{t}_i^0 \quad\text{and}\quad \beta_i[s] = \mathbf{a}_i[s]\cdot\mathbf{b}_i^0 . \tag{11}$$

The scalar values $\alpha_i[s]$ and $\beta_i[s]$ are quantities which do not depend on the current pose and can be precomputed statically for a rigged mesh $\mathcal{R}$. We call them *deform-factors*.

4.3 Numerical stability and bandwidth optimization

Recall that the first three coordinates of $\mathbf{a}_i[s]$ are actually the gradient of the weight function $\omega_i[s]$. Since the weights partition unity, i.e. the sum $\sum_{s\in\mathcal{S}}\omega_i[s] = 1$ is constant, the sum of the weight gradients is zero. Therefore, by Eq. (11) we have:

$$\forall i,\ \sum_{s\in\mathcal{S}}\alpha_i[s] = 0 \quad\text{and}\quad \sum_{s\in\mathcal{S}}\beta_i[s] = 0. \tag{12}$$

This indicates that the summation in Eq. (10) can contain terms with much larger magnitude than the result, potentially leading to numerical cancellation errors, especially since GPU computations are typically performed in single precision. The problem occurs in practice, as shown in Fig. 4.

As follows from Eq. (12), any constant vector $\mathbf{c}$ can be added to all $\mathbf{q}_i^t[s]$ without changing the result in Eq. (10). We

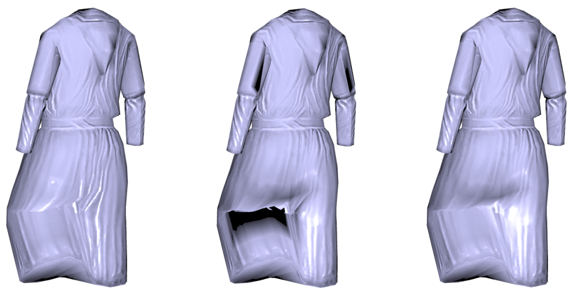

Figure 5: *Our method enhances defects of the input (middle). With a simple modification, we can make our method (right) as robust as the standard algorithm (left). See also attached video.*

use $\mathbf{c} = -\mathbf{q}_i^t[s_0]$, where s_0 is an arbitrarily chosen $s\in\mathcal{B}_i$ (e.g., the first one stored in $\mathcal{B}_i$). This nullifies the first term of the summation and significantly reduces the magnitude of the remaining terms. It also removes the need to store one of the $N_{\max}$ deform-factor pairs per vertex. Eq. (10) becomes:

$$\begin{aligned}\mathbf{t}_i^t &= \mathbf{T}_i^t\mathbf{t}_i^0 + \sum_{s\in\mathcal{B}_i\setminus\{s_0\}} \alpha_i[s]\left(\mathbf{q}_i^t[s] - \mathbf{q}_i^t[s_0]\right) \\ \mathbf{b}_i^t &= \mathbf{T}_i^t\mathbf{b}_i^0 + \sum_{s\in\mathcal{B}_i\setminus\{s_0\}} \beta_i[s]\left(\mathbf{q}_i^t[s] - \mathbf{q}_i^t[s_0]\right)\end{aligned} \tag{13}$$

As an optimization, we can reuse the values computed in Eq. (13) to transform the vertex coordinates. Eq. (1) becomes:

$$\begin{aligned}F_i^t(\mathbf{p}^0) &= \sum_{s\in\mathcal{B}_i}\omega_i[s]\,\mathbf{q}_i^t[s] = \\ &= \mathbf{q}_i^t[s_0] + \sum_{s\in\mathcal{B}_i\setminus\{s_0\}}\omega_i[s]\left(\mathbf{q}_i^t[s] - \mathbf{q}_i^t[s_0]\right).\end{aligned} \tag{14}$$

4.4 Handling flipped normals

The method described above reproduces normals that are similar to the geometric normals of the deformed mesh, which was our objective. In most cases, this improves the shading, making it more communicative of the actual deformation taking place. However, the skinning deformation may sometimes make some mesh triangles degenerate (nearly collapsed to a line), and the orientation of some triangles may occasionally flip. This can happen when the bone weights vary too rapidly over $\mathcal{R}$, for example when vertices of an extremely small triangle are assigned to significantly different weights. Since a vertex is shared by several triangles, the effect of the normal of a flipped face can "bleed" to neighboring faces, causing a noticeable artifact (see Fig. 5), even when the folded or collapsed triangle itself is not very visible in the deformed mesh.

Strictly speaking, these artifacts are caused by inconsistencies in the input, and not by a defect of our method. However, the traditional approximation (Eq. (3)), being oblivious to non-rigid deformations, is *ipso facto* more resistant to such inconsistencies; this simplifies the task of rigging the meshes (e.g. by artists or by automatic algorithms). Moreover, it can be desirable to adopt the new method directly to existing rigged models designed for (and tested with) the approximated lighting.

If deemed appropriate, we can make our method as robust as the approximate one, compromising only very little of its accuracy in non-degenerate cases. It is sufficient to artificially bound the magnitude of the second summation term in both Eqs. (13) to a maximal value c, where $c < 1$ is a constant (i.e., if the vector has length greater than c, it is scaled down to c).

In our experiments, we picked $c = 0.9$. While this value is sufficient for a complete or nearly complete correction of the tangent directions, it prevents their inversion in the degenerate cases. See Fig. 5 for an example.

4.5 Adaptation to DQS

The presented formulation can be easily adapted to serve as a good approximation for normals transformed by DQS. The same deform factors $\alpha[s], \beta[s]$ are used. The first term of Eq. (13) is rotating the vectors $\mathbf{t}_i^0$ and $\mathbf{b}_i^0$ by the blended dual quaternion (instead of the blended matrix $\mathbf{T}_i^t$). In the second term of Eq. (13), the $\mathbf{q}_i^t[s]$ are computed by rigidly transforming $\mathbf{p}_i^0$ with the dual quaternion defined for bone s in the current pose t.

5 Implementation recipe

In this section, we summarize our algorithm. A C++/GLSL implementation is provided in the additional material.

Preprocessing. Given a rigged mesh $\mathcal{R}$ with the following per-vertex attributes: vertex position $\mathbf{p}_i^0 \in \mathbb{R}^3$, normal $\mathbf{n}_i^0$, texture coordinates (u_i, v_i), a set of (up to) $N_{\max}$ bone indices $\mathcal{B}_i$, and $N_{\max}$ weights $\omega_i[s]$ $(s \in \mathcal{B}_i)$.

1. Compute and store per-vertex tangent directions $\mathbf{t}_i^0$, $\mathbf{b}_i^0$. Enforce both vectors to be orthogonal to $\mathbf{n}_i^0$ (Eq. (9)).
2. Compute temporary per-face vectors $\hat{\mathbf{a}}[s]$, solving the system in Eq. (5), for each bone $s \in \mathcal{S}$ that has nonzero influence on any vertex of the processed face. Accumulate into incident vertices, finding temporary per-vertex vectors $\mathbf{a}_i[s]$. If $\mathbf{a}_i[s] \neq 0$ but $s \notin \mathcal{B}_i$, add s to $\mathcal{B}_i$ with weight $\omega_i[s] = 0$. Discard supernumerary nonzero vectors (see Sec. 3.2).
3. Compute and store, per-vertex, $(N_{\max} - 1)$ pairs $(\alpha_i[s], \beta_i[s])$, for all $s \in \mathcal{B}_i$ except one, using Eq. (11). Discard temporaries.

At rendering time. In the vertex shader, compute transformed tangent directions $\mathbf{t}_i^t, \mathbf{b}_i^t$ in pose space with Eq. (13) and transformed normals $\mathbf{n}_i^t$ as their cross product. The rest of the rendering pipeline is unaffected.

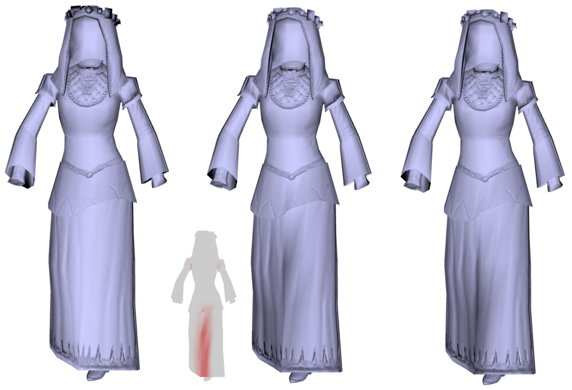
Figure 6: *An animation with Linear Blend Skinning: the standard method (left), our method (middle) and the ground truth (right). See also attached video.*

6 Results and evaluation

We evaluate the achieved visual quality and the total costs in terms of memory and computation time of our technique.

Visual quality. We ran our algorithm on several low-poly, bump-mapped, rigged models and animations taken from games, testing both the LBS and the DQS cases. Several images throughout this paper show the results, compared with the standard approximation of Eq. (3). Still images are only partly meaningful, and the reader is kindly referred to the attached videos (each image in this paper has a corresponding short video sequence, and we attach several additional examples). All the results can also be inspected interactively using the attached implementation, which embeds the datasets used in this paper.

The visual comparisons show that our accurate lighting is substantially more effective than the standard approximation in conveying the animated shape. The improvement is usually more evident in the less rigid deformations areas.

We also wish to indicatively assess the combined effects of the assumptions and small approximations made by our

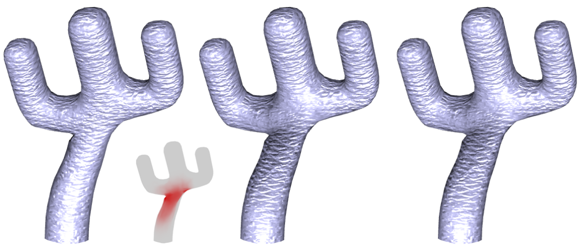
Figure 7: *An animation of a simple character with Linear Blend Skinning, which uses $N_{\max} = 2$. Same caption as Fig. 6. See also attached video.*

	Per-vertex operations						Per-vertex bandwidth (in bytes)		
	LBS			DQS					
N_{max}	2	3	4	2	3	4	2	3	4
Approximation	63	75	87	106	118	120	12	18	24
Ours (overhead)	+12	+30	+48	+57	+90	+133	+8	+16	+24
Naive (overhead)	+67	+85	+103	-	-	-	+24	+36	+48
Ours (relative)	+19%	+40%	+55%	+54%	+76%	+111%	+67%	+89%	+100%
Naive (relative)	+106%	+113%	+118%	-	-	-	+200%	+200%	+200%

Table 1: *Comparison of operation counts (*multiply-and-add *operations) and bandwidth (bytes) required per-vertex.* Approximation *stands for the approximate normal computation (Eq. (3)),* Naive *stands for the direct application of the exact Jacobian, as described in Sec. 3, and* Ours *denotes our method. At the bottom, we report the normalized results compared with the commonly used approximation. Refer to the supplemental material for a detailed explanation of the counts.*

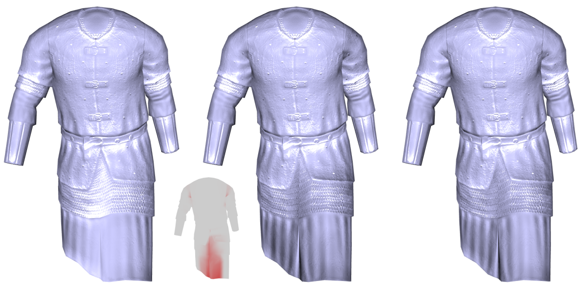

Figure 8: *An animation with Dual Quaternion Skinning. Same caption as Fig. 6. See also attached video.*

system, which are: considering the weights as linearly interpolated inside faces (Sec. 3.1), discarding supernumerary weights, lumping averaged per-face gradients to vertices (Sec. 3.2), capping the correction to prevent possible normal flipping (Sec. 4.4), and the additional approximation for DQS (Sec. 4.5). To do so, we compared with a "ground truth", which consists of a model deformed off-line for a given pose (with LBS or DQS), over which we statically recompute normal and tangent directions from the deformed geometry. The results, shown in Fig. 6, 7, 8, and in the attached videos, demonstrate that our method gets much closer to this ground truth than the commonly used approximation.

The datasets of all the examples in this paper and in the attached videos use $N_{\max} = 4$, with the exception of the cactus in Fig. 7, which uses $N_{\max} = 2$.

Operation count and bandwidth. We compare in Table 1 the number of operations and bytes required at run time by our algorithm on a per-vertex basis. We compare with the commonly used approximation of Eq. (3), and also with the direct computation of Eq. (7). We compare the methods using values of $N_{\max}$ between 2 (typical for games on mobile devices) and 4 (the value supported by most game engines). In LBS, the overhead on operation count, compared with the standard approximation, is negligible for a low $N_{\max}$, and it caps at 55%. For DQS, adopting our solution more or less doubles the computational costs. The bandwidth overhead is very limited, requiring between 8 to 24 additional bytes per vertex (the representation of animations is unaffected). The overhead relative to the commonly used approximation ranges between 100% ($N_{\max} = 4$) and 67% ($N_{\max} = 2$). In a typical context where positions, normals, texture coordinates, color, skinning weights, and tangent directions are sent per-vertex, the overall bandwidth overhead is less than 29% ($N_{\max} = 4$).

7 Discussion

We demonstrated that accurate normals can be affordably computed, for LBS and DQS, eliminating the illumination discrepancies introduced by the ubiquitously adopted loose approximation. Our solution can be easily integrated into existing rendering engines.

The balance between visual quality and consumed memory and processing resources has kept shifting throughout the history of video games, as testified by the continuous increase in poly-counts, texture resolutions, number of texture sheets, shader program lengths, amount of per-vertex data, and so on. Ten years ago, when skinning techniques were already the established standard to handle deformable bodies, the rather loose approximation of the normal transformations was fully in line with other compromises. Since then, it has not changed, possibly due to the difficulties highlighted in Sec. 3.3 and addressed by this paper, and by now this approximate technique appears outdated. Note that the approximation error, differently from many other cases, does not get better by simply refining the meshes.

For these reasons, we believe that our technique may have a significant impact in the near future of interactive applications.

Implementation and videos available at the project page:
http://vcg.isti.cnr.it/deformFactors

Acknowledgments

Models, textures and animations are from Mount&Blade Warband, used with permission from TaleWorlds Entertainment. We thank the anonymous reviewers for their insightful comments, Alec Jacobson and Ladislav Kavan for the useful discussions and Emily Whiting for narrating the accompanying video. This work was supported in part by the ERC grant iModel (StG-2012- 306877), by EU FP7 project ICT FET "Harvest4D"(G.A. no. 323567), by an SNSF award 200021_137879 and a gift from Adobe Research.

References

[BJ07] Barbič J., James D.: Time-critical distributed contact for 6-dof haptic rendering of adaptively sampled reduced deformable models. In *Proceedings of the 2007 ACM SIGGRAPH/Eurographics symposium on Computer animation* (2007), Eurographics Association, pp. 171–180. 1

[CdGDS13] Crane K., de Goes F., Desbrun M., Schr oder P.: Digital geometry processing with discrete exterior calculus. In *ACM SIGGRAPH 2013 courses* (New York, NY, USA, 2013), SIGGRAPH '13, ACM. 4

[DSP06] Der K. G., Sumner R. W., Popović J.: Inverse kinematics for reduced deformable models. *ACM Trans. Graph. 25*, 3 (July 2006), 1174–1179. 1

[FO06] Forstmann S., Ohya J.: Fast skeletal animation by skinned arc-spline based deformation. *EG 2006 Short Papers* (2006), 1–4. 1

[HYC*05] Hyun D.-E., Yoon S.-H., Chang J.-W., Seong J.-K., Kim M.-S., Jüttler B.: Sweep-based human deformation. *The Visual Computer 21*, 8-10 (2005), 542–550. 1

[JBK*12] Jacobson A., Baran I., Kavan L., Popović J., Sorkine O.: Fast automatic skinning transformations. *ACM Trans. Graph. 31*, 4 (2012), 77:1–77:10. 1

[JS11] Jacobson A., Sorkine O.: Stretchable and twistable bones for skeletal shape deformation. *ACM Trans. Graph. 30*, 6 (2011), 165:1–165:8. 1

[JT05] James D. L., Twigg C. D.: Skinning mesh animations. *ACM Trans. Graph. 24*, 3 (July 2005), 399–407. 1

[KCZO07] Kavan L., Collins S., Zara J., O'Sullivan C.: Skinning with dual quaternions. In *Proc. I3D* (2007), pp. 39–46. 1

[KJP02] Kry P. G., James D. L., Pai D. K.: Eigenskin: Real time large deformation character skinning in hardware. In *Proceedings of the 2002 ACM SIGGRAPH/Eurographics Symposium on Computer Animation* (New York, NY, USA, 2002), SCA '02, ACM, pp. 153–159. 1

[KSO10] Kavan L., Sloan P.-P., O'Sullivan C.: Fast and efficient skinning of animated meshes. *Comput. Graph. Forum 29*, 2 (2010), 327–336. 1

[LS10] Landreneau E., Schaefer S.: Poisson-based weight reduction of animated meshes. *Comput. Graph. Forum 29*, 6 (2010), 1945–1954. 2

[MG03] Mohr A., Gleicher M.: Building efficient, accurate character skins from examples. *ACM Trans. Graph. 22*, 3 (July 2003), 562–568. 1

[MMG06] Merry B., Marais P., Gain J.: Normal transformations for articulated models. In *ACM SIGGRAPH 2006 Sketches* (2006), p. 134. 3

[MTLT88] Magnenat-Thalmann N., Laperrière R., Thalmann D.: Joint-dependent local deformations for hand animation and object grasping. In *Proc. Graphics Interface* (1988). 1

[RJ07] Rivers A. R., James D. L.: FastIsm: Fast lattice shape matching for robust real-time deformation. *ACM Trans. Graph. 26*, 3 (July 2007).

[VBG*13] Vaillant R., Barthe L., Guennebaud G., Cani M.-P., Rohmer D., Wyvill B., Gourmel O., Paulin M.: Implicit skinning: Real-time skin deformation with contact modeling. *ACM Trans. Graph. 32*, 4 (2013), 125:1–125:12. 1

[WPP07] Wang R. Y., Pulli K., Popović J.: Real-time enveloping with rotational regression. *ACM Trans. Graph. 26*, 3 (2007), 73. 1

[YSZ06] Yang X., Somasekharan A., Zhang J. J.: Curve skeleton skinning for human and creature characters. *Computer Animation and Virtual Worlds 17*, 3-4 (2006), 281–292. 1