This is the author's accepted version of the article:

Wrinkling Coarse Meshes on the GPU
by Jörn Loviscach
Computer Graphics Forum 25(3), pp. 467-476
(Proceedings of Eurographics 2006)

The definitive version is available at diglib.eg.org and www.blackwell-synergy.com.

EUROGRAPHICS 2006 / E. Gröller and L. Szirmay-Kalos
(Guest Editors)

Volume 25 (2006), Number 3

Wrinkling Coarse Meshes on the GPU

J. Loviscach

Fachbereich Elektrotechnik und Informatik, Hochschule Bremen, Germany

Abstract

The simulation of complex layers of folds of cloth can be handled through algorithms which take the physical dynamics into account. In many cases, however, it is sufficient to generate wrinkles on a piece of garment which mostly appears spread out. This paper presents a corresponding fully GPU-based, easy-to-control, and robust method to generate and render plausible and detailed folds. This simulation is generated from an animated mesh. A relaxation step ensures that the behavior remains globally consistent. The resulting wrinkle field controls the lighting and distorts the texture in a way which closely simulates an actually deformed surface. No highly tessellated mesh is required to compute the position of the folds or to render them. Furthermore, the solution provides a 3D paint interface through which the user may bias the computation in such a way that folds already appear in the rest pose.

Categories and Subject Descriptors (according to ACM CCS): I.3.7 [Computer Graphics]: Animation, I.3.7 [Computer Graphics]: Color, shading, shadowing, and texture

1. Introduction

Most 3D character animation—be it for film production or for interactive applications such as games or simulations—demands a realistic depiction of the actors. Garments and their interaction with an actor's body contribute greatly to the perceived level of realism. However, accurate physical simulation of cloth still incurs high costs in computation and design: The resulting stiff high-dimensional differential equations together with the collision detection and handling are difficult to treat robustly. The results are hard to guess in advance for the user; intuitive parameters such as the density of wrinkles per inch are not easily available.

This work describes a fully GPU-based real-time approach that addresses these problems, see Figure 1. The method takes an animated coarse mesh as input, evaluates the deformation of this surface due to the animation ("Skin"), determines the local strength and direction of the wrinkles ("Crush"), aligns them ("Relax"), and renders them via shading of the mesh's original triangles ("Render"), see Figure 2. This approach is not based on physical dynamics, but on the kinematical preservation of length through buckling. The main contributions of this paper are:

- a robust and controllable method to determine a highly detailed, globally consistent wrinkle field from arbitrary geometric deformations of a mesh with no precomputed or predesigned folds,
- a method to generate bump mapping and texture deformation on a coarse mesh from non-static height fields,
- a method to paint rest-pose wrinkles, which interact plausibly with the wrinkles generated from an animation.

The input animation can be of any kind, interactive or prerecorded, if the resulting vertex positions and normals are accessible in a shader. The wavelength of the wrinkles as well as their height profile (such as sinusoidal or accordion-like) can be controlled interactively. A real-time prototype was developed using Microsoft® Managed DirectX® 9.0c and HLSL for Shader Model 3.0. However, the method is also applicable to offline rendering, for instance to create wrinkles in 3D modeling software, to design morph targets, or to provide a starting point for further editing.

This paper is structured as follows: Section 2 provides an overview over related work. Section 3 describes the preparation of the mesh. Its deformation and the determination of the strength and direction of the wrinkles is covered in Section 4. The generation and global alignment of the wrinkle field is described in Section 5. Section 6 addresses the ren-

© The Eurographics Association and Blackwell Publishing 2006. Published by Blackwell Publishing, 9600 Garsington Road, Oxford OX4 2DQ, UK and 350 Main Street, Malden, MA 02148, USA.

Figure 1: *The described method adds plausible wrinkles to an animated coarse mesh (original: upper left). The wavelength of the wrinkles can be chosen freely (upper diagonal).*

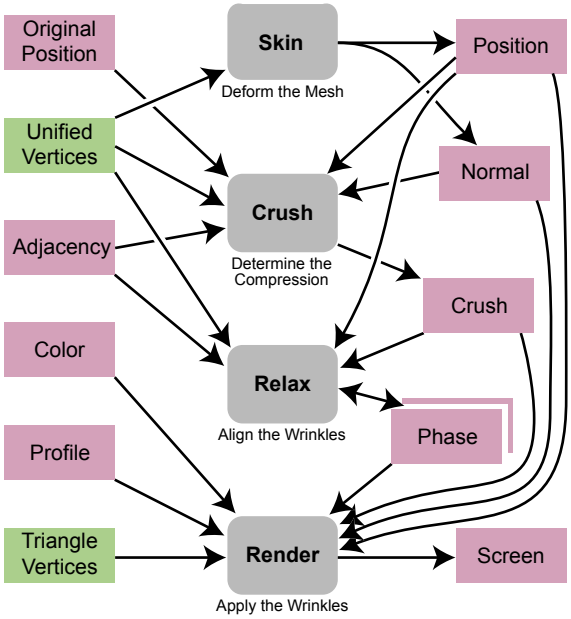

Figure 2: *Four pairs of vertex and pixel shaders (rounded rectangles) form the pipeline to generate and render the wrinkles. In the intermediate steps, data is read from vertex buffers and read from and written to off-screen pixel buffers.*

dering of the wrinkles. The method employed to create and incorporate rest-pose wrinkles is presented in Section 7. Results are given in Section 8; Section 9 concludes the paper.

2. Related work

Cloth simulation and rendering has been an active field for 20 years. An up-to-date survey is given by Magnenat-Thalmann and Volino [MTV05]. Many cloth simulation methods that aim at high realism employ differential equations inspired by physics. However, geometric methods have always presented themselves as an alternative, due to their speed and controllability.

Kunii and Gotoda [KG90] describe primitives for wrinkle lines based on singularity theory. Ng and Grimsdale [NG95] create fold lines by geometric rules and deform a mesh accordingly. Combaz and Neyret [CN02] describe a modeling application, in which the user may paint the amount and direction of surface expansion and set the desired wavelength and the amount of regularity. The mesh grows and folds according to the input. Larboulette and Cani [LC04] use a curve on a surface, along which the mesh is contracted with length preservation. In a similar vein, Wang et al. [WWY06] fold the neighborhood of a curve using a physical simulation and locally refining the mesh.

Kang et al. [KCCL01] animate a regularly structured coarse mesh and place fine-scale vertices using cubic spline curves. These splines oscillate to have constant length. Oshita and Makinouchi [OM01] deform PN triangles in such a way that their curved edges retain their length. Kono and Genda [KG03] control the bulging of muscle-like ribbons to preserve length.

Hadap et al. [HBVMT99] determine a bump map for a coarse mesh from a wrinkle pattern, which is prepared as a texture by the user. The local amplitude is computed interactively with a method inspired by the preservation of area. Bando et al. [BKN02] create bump-mapped fine-scale wrinkles by interpolating a user-specified direction field. They also produce large-scale wrinkles from Bézier curves drawn in texture space. The wrinkle strength is determined from the contraction perpendicular to a given fold. Cutler et al. [CGW*05] allow the user to edit wrinkle curves for reference poses. The wrinkle pattern for an arbitrary pose is created by blending the prepared patterns according to the similarity of the local geometry, that is: the lengths of and angles between the vectors from a vertex to its neighbors.

Herman [Her01] computes folding patterns for hundreds of reference poses using an offline simulation. For fast interaction, he blends those patterns according to the similarity of the current pose to the reference poses. Kang and Cho [KC02] apply fast but dissipative physics to a coarse mesh with little contraction resistance and then use the result to control a fine mesh with high contraction resis-

© The Eurographics Association and Blackwell Publishing 2006.

tance, which leads to strong folds. Cordier and Magnenat-Thalmann [CMT05] animate a coarse mesh and determine the deformation of a fine mesh from the coarse one using linear regression. This linear regression is trained with results obtained from offline cloth simulations. Decaudin et al. [DJW*06] employ a coarse control mesh to generate diamond and twist buckling shapes on the cloth.

Wrinkles and folds controlled by the amount of mesh deformation can also be found in commercial products: NewTek LightWave 3D [New05] can compute a "Tension Map." The "SparseMorphTargets Sample" from the DirectX 10 SDK [Mic05] employs geometry shaders to determine the triangles' area change under animation.

Many authors aim at improving bump mapping to incorporate parallax effects. Kankeko et al. [KTI*01] introduce the simple, yet effective "parallax mapping." Here, the surface is treated as weakly curved, from which easily follows a correction for the texture coordinates. Welsh [Wel04] adds a limiting term to this method to counteract distortion for grazing angles. He also treats curved surfaces, which, however, still require a uniform texture mapping. At a much higher computational cost, ray casting in the pixel shader offers better image quality, see for instance Tatarchuk [Tat05]. This idea is taken further by Policarpo et al. [POC05], who extend the ray casting process to render curved patches. Wang et al. [WWT*03] precompute ray-surface intersection data parameterized by location, direction, and the curvature of the base. The result is reduced to a set of textures via PCA.

3. Mesh preparation

The animated mesh as contained in a DirectX .x file is described through a vertex list and an triangle list. Two of the four computation steps have to retrieve the neighbors of a given vertex. This operation, however, is not yet supported by PC-based graphics hardware. Thus, the adjacency data have to be collected upfront and stored in a buffer.

3D objects employed for character animation often are mapped onto the surface with seams, see Figure 3. The typical method to create such seams is to use multiple copies of vertices with the same positions but different texture coordinates. For the simulation, such duplicated vertices have to be merged again. This list of unified vertices and their adjacency data form the basis of the simulation, see Figure 2.

On initialization, the adjacency data for the unified vertices are stored in a buffer of size 512×512. (This buffer is hugely oversized for one single object; however, the data of dozens of objects can be stored in the same buffer.) In standard writing order, the ID numbers of the neighbors of vertex 1 are stored first, then those of vertex 2, and so on. Whenever the space in a row is too tight to add the next vertex' neighbors completely to that row, the placement continues in the next row with no wrapping, see Figure 4. This

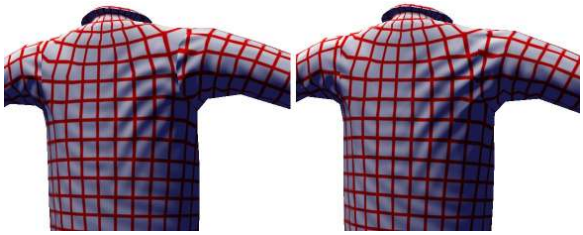

Figure 3: *Whereas the texture should show a discontinuity at seams (see the shoulders), the wrinkle field should not. Thus, vertices duplicated to form seams (left) have to be treated as one (right).*

causes a tiny memory overhead but greatly simplifies the access, which is time-critical. (This overhead could be eliminated if the graphics cards allowed a one-dimensional buffer that is thousands of pixels wide.)

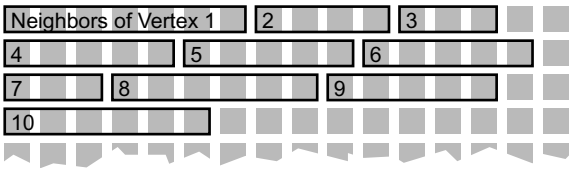

Figure 4: *To allow easy access, the neighbor data are not wrapped from one line to the next.*

To work with the adjacency data, the (unique) vertices carry an additional attribute consisting of three numbers: their ID, their number of neighbors, and the starting address of their data in the adjacency buffer.

4. Deformation

The deformation part of the computation consists of two steps: First, the positions of the surface's vertices are animated ("Skin" in Figure 2), a process that results in new positions and new vertex normals. Second, the deformation around every vertex is determined by linear approximation. Per vertex, this leads to a strength value and a vector that describe the direction of strongest compression ("Crush").

4.1. Skinning

The wrinkle generation can process any vertex-based animation, be it generated through bones, morphing (aka mesh blending), or physical simulation. Due to the prevalence of bone-based animations in current real-time applications, the prototype, too, relies on a bone skeleton to which the mesh is attached by soft skinning. Loading animations and setting a matrix palette accordingly is mostly realized through standard DirectX functionality. The evaluation of the positions and normals is handled in a vertex shader.

© The Eurographics Association and Blackwell Publishing 2006.

In the prototype, every vertex can be influenced by at most four ($k = 1, \ldots, 4$) bone transformations. The attributes of the i-th vertex comprise the indices $m(i,k)$ of these affine transfomations $(M_m, \mathbf{c}_m)$—where M_m is a 3×3 matrix and $\mathbf{c}_m$ is a translation vector—and the corresponding bone weights $\alpha(i,k)$. The post-deformation position $\mathbf{x}_i$ of the i-th vertex is determined from the rest-space position $\mathbf{x}_i$ via

$$\mathbf{x}_i' = \sum_{k=1}^{4} \alpha(i,k)(M_{m(i,k)}\mathbf{x}_i + \mathbf{c}_{m(i,k)}).$$

Employing a common approximation from game development, we obtain the non-normalized post-deformation normal vector $\mathbf{n}'$ from the original $\mathbf{n}$ by

$$\mathbf{n}_i' = \sum_{k=1}^{4} \alpha(i,k) M_{m(i,k)}\mathbf{n}_i.$$

Homogeneous coordinates unify both computations.

The input to these computations is the list of unique vertices: If two vertices share the same position in space to allow texture seams, they will be deformed equally, so we need to only cover one vertex of every such group. For this and the next two steps ("Skin," "Crush," and "Relax" in Figure 2), the rendering mode is set to create points instead of triangles. All vertices are mapped to single pixels in two off-screen floating-point buffers of size 256×256: one for position data, one for normal data. Writing to both buffers at the same time requires Multiple Render Target functionality.

4.2. Computation of the compression

On startup, the original positions of the vertices are stored in a buffer similar to the buffer for the deformed vertices. By looking at a vertex and its neighbors before and after deformation, one can deduce a local linear approximation of the deformation.

Let $\mathbf{x}_0$ be a the position and $\mathbf{n}$ the normal of a certain vertex in the rest pose, and likewise $\mathbf{x}_i$ the positions of its direct neighbors $i = 1, \ldots, N$. Let $\mathbf{x}_0'$, $\mathbf{n}'$, and $\mathbf{x}_i'$ be the corresponding quantities after deformation. Then we create orthonormal frames in both the rest-pose and the post-deformation tangent spaces through

$$\mathbf{t} = \text{normalize}(\mathbf{x}_1 - \mathbf{x}_0 - ((\mathbf{x}_1 - \mathbf{x}_0)\cdot\mathbf{n})\mathbf{n}), \quad \mathbf{b} = \mathbf{n} \times \mathbf{t},$$
$$\mathbf{t}' = \text{normalize}(\mathbf{x}_1' - \mathbf{x}_0' - ((\mathbf{x}_1' - \mathbf{x}_0')\cdot\mathbf{n}')\mathbf{n}'), \quad \mathbf{b}' = \mathbf{n}' \times \mathbf{t}'.$$

The difference vectors from the central vertex 0 to its neighbors lead to tangent-space vectors

$$\mathbf{v}_i = \begin{pmatrix} \mathbf{t}\cdot(\mathbf{x}_i - \mathbf{x}_0) \\ \mathbf{b}\cdot(\mathbf{x}_i - \mathbf{x}_0) \end{pmatrix}, \quad \mathbf{v}_i' = \begin{pmatrix} \mathbf{t}'\cdot(\mathbf{x}_i' - \mathbf{x}_0') \\ \mathbf{b}'\cdot(\mathbf{x}_i' - \mathbf{x}_0') \end{pmatrix}.$$

We seek a 2×2 matrix M that optimally approximates the local deformation in the sense that it minimizes the sum of quadratic errors

$$\sum_{i=1}^{N} |\mathbf{v}_i' - M\mathbf{v}_i|^2. \tag{1}$$

This leads to a multidimensional linear regression

$$M = \left(\sum_{i=1}^{N} \mathbf{v}_i' \otimes \mathbf{v}_i\right)\left(\sum_{i=1}^{N} \mathbf{v}_i \otimes \mathbf{v}_i\right)^{-1},$$

where $\otimes$ denotes the tensor product.

To find the strength r and a direction $\mathbf{k}$ of maximum compression in pre-deformation space, we have to minimize

$$\frac{|M\mathbf{v}|^2}{|\mathbf{v}|^2} = \frac{\mathbf{v}\cdot M^{\mathrm{T}}M\mathbf{v}}{|\mathbf{v}|^2}$$

for $\mathbf{v} \in \mathbb{R}^2$. Hence, r^2 is the smaller of the two eigenvalues of $M^{\mathrm{T}}M$, and $\mathbf{k}$ can be found as $(\mathbf{t}, \mathbf{b})\mathbf{v}$, where $\mathbf{v}$ is a corresponding eigenvector $\in \mathbb{R}^2$. We require $|\mathbf{k}| = 1$ so that this vector is determined up to its sign. This ambiguity will be taken care of in later phases.

The strength of the compression is of less interest than the amplitude of the wrinkles generated from it. Figure 5 shows the geometric relation between the compression and the height for a accordion-type wrinkle. (Computing the precise arc length of *sinusoidal* wrinkles turned out to be an unnecessary burden.) On expansion, the wrinkle amplitude h should be zero, so that:

$$h = \frac{W}{4}\sqrt{\max(1 - r^2, 0)}, \tag{2}$$

where W is the wrinkle's wavelength, which we regard to be a constant in pre-deformation space.

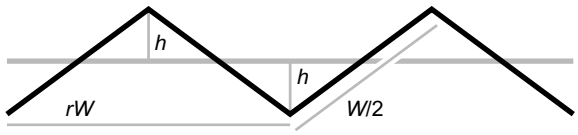

Figure 5: *For a wrinkle with accordion profile, the relation between compression and height can be computed easily.*

All of these computations can be handled in a pixel shader, which reads (among others) the buffers containing the original positions, the deformed positions, and the deformed normals, and writes its result (direction, amplitude) to another off-screen buffer ("Crush"), in which again one pixel represents one vertex.

5. Wrinkle field

The "wrinkle field" is a height field that is computed as though it would deform the polygonal mesh along the direction of the local normal vector. The later rendering will *not* actually execute such a displacement mapping but only simulate its visual effects. The wrinkle field is not stored as a regular texture, but computed on the fly from amplitude and direction data stored per vertex. Near every vertex, the wrinkle field is represented by a plane wave. To align the waves of neighboring vertices, a relaxation process is employed that determines the wave's phase at every vertex.

© The Eurographics Association and Blackwell Publishing 2006.

5.1. Computing the displacement

We assume that the wrinkle width W is constant in rest-pose space. The geometric transformation of the vertices will then automatically lead to an appropriate compression of the wrinkles. The deformation computations lead to a wrinkle amplitude h and a tangential normalized direction vector $\mathbf{k}$ per vertex. In rest-pose space, we model the displacement (that is: the local height value) at all points $\mathbf{x}$ near a vertex A by the plane wave

$$h(\mathbf{x}) = h_A \cos\left(\frac{2\pi}{W}\mathbf{k}_A \cdot (\mathbf{x}-\mathbf{x}_A) + \phi_A\right), \qquad (3)$$

where h_A is the amplitude, $\mathbf{k}_A$ is the direction, ϕ_A is the phase at the vertex A, and $\mathbf{x}_A$ denotes its position, see Figure 6. (Here we employ a sinusoidal profile. Subsection 6.3 addresses how generalize this.)

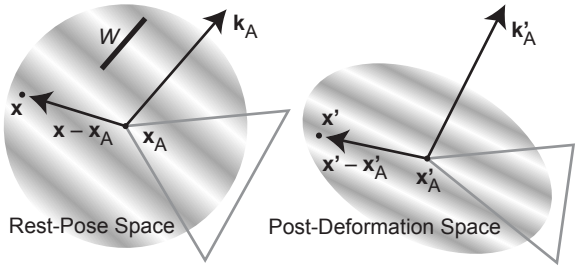

Figure 6: *The wrinkle width W is constant in rest-pose space. The actual computations, however, are carried out in post-deformation space.*

To avoid conversions from rest-pose space to post-deformation space, the actual computation is executed in the latter. Thus, we have to convert $\mathbf{k}$ from rest-pose space to a corresponding $\mathbf{k}'$ in post-deformation space. The phase of the wave in Equation 3 must not change, see Figure 6:

$$\mathbf{k}'_A \cdot (\mathbf{x}' - \mathbf{x}'_A) = \mathbf{k}_A \cdot (\mathbf{x} - \mathbf{x}_A). \qquad (4)$$

We have already determined a 2×2 matrix M to approximate the deformation in 2D tangent space around every vertex, see Equation 1. A full 3D version of this linear transformation can be found via the matrix L defined through

$$L\mathbf{x} = (\mathbf{t}', \mathbf{b}')M \begin{pmatrix} \mathbf{t}\cdot\mathbf{x} \\ \mathbf{b}\cdot\mathbf{x} \end{pmatrix}.$$

Therefore, in the vicinity of vertex A the following holds:

$$\mathbf{x}' - \mathbf{x}'_A \approx L(\mathbf{x} - \mathbf{x}_A),$$

Thus, Equation 4 leads to the choice

$$\mathbf{k}'_A = (L^{-1})^{\mathrm{T}} \mathbf{k}_A.$$

The corresponding field of per-vertex vectors $\mathbf{k}'$ is computed and stored by the "Crush" shader described in Subsection 4.2.

To form a continuous height field, the height is interpolated linearly over every triangle ABC of the mesh:

$$\begin{aligned} h(\mathbf{x}') \;=\; & \alpha h_A \cos\left(\frac{2\pi}{W}\mathbf{k}'_A \cdot (\mathbf{x}' - \mathbf{x}'_A) + \phi_A\right) \\ & + \beta h_B \cos\left(\frac{2\pi}{W}\mathbf{k}'_B \cdot (\mathbf{x}' - \mathbf{x}'_B) + \phi_B\right) \\ & + \gamma h_C \cos\left(\frac{2\pi}{W}\mathbf{k}'_C \cdot (\mathbf{x}' - \mathbf{x}'_C) + \phi_C\right), \end{aligned} \qquad (5)$$

where α, β, and γ are the barycentric coordinates of $\mathbf{x}'$.

5.2. Relaxation

Whereas the amplitude h_A and the direction $\mathbf{k}'_A$ of the wrinkle field have already been determined for every vertex, the phase ϕ_A is still unknown. One has to be careful in selecting values for the phase because otherwise the surface appears crumpled, see Figure 7.

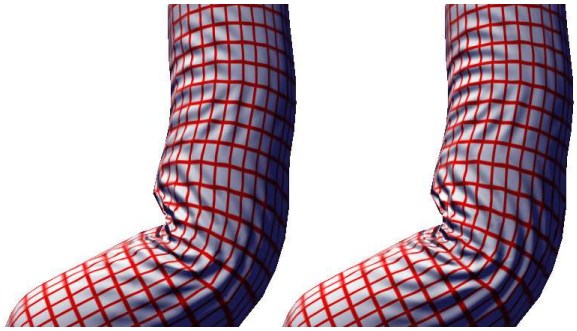

Figure 7: *The local plane waves do not match up if the phase value per vertex is set arbitrarily (left). To form consistent wrinkles, a relaxation process is employed (right).*

The ϕ data are initialized with random numbers. At every time step, the ϕ of every vertex is changed slightly toward a value that reduces the misfit between the plane wave at the vertex and those at its neighbors. Current graphics hardware does not allow reading from and writing into the same buffer within the same shader. Hence, two buffers for the ϕ data are used alternatingly for reading and writing, see Figure 2.

We want to adjust the ϕ values in such a way that halfway between a vertex 0 and its neighbor n the corresponding plane waves coincide. This means

$$\cos\left(\frac{2\pi}{W}\mathbf{k}'_n \cdot (\mathbf{p} - \mathbf{x}'_n) + \phi_n\right) \approx \cos\left(\frac{2\pi}{W}\mathbf{k}'_0 \cdot (\mathbf{p} - \mathbf{x}'_0) + \phi_0\right)$$

for $\mathbf{p} = (\mathbf{x}'_n + \mathbf{x}'_0)/2$. The signs of the vectors $\mathbf{k}'_0$ and $\mathbf{k}'_n$ have not been determined uniquely. Due to the symmetry of the cosine, flipping one of these does not affect the resulting height if also the sign of the corresponding ϕ is changed. To directly compare the phases, however, we have to take the signs into account, and this choice has to be consistent

© The Eurographics Association and Blackwell Publishing 2006.

between adjacent vertices. One can look at the sign of the dot product $\mathbf{k}'_n \cdot \mathbf{k}'_0$ to determine whether $\mathbf{k}'_n$ should be flipped to conform to $\mathbf{k}'_0$. Hence, we introduce the phases at the vertex 0 and its neighbor n as:

$$\begin{aligned}\psi_0 &= \frac{2\pi}{W}\mathbf{k}'_0 \cdot (\mathbf{p} - \mathbf{x}'_0) + \phi_0, \\ \psi_n &= \operatorname{sgn}(\mathbf{k}'_n \cdot \mathbf{k}'_0)\left(\frac{2\pi}{W}\mathbf{k}'_n \cdot (\mathbf{p} - \mathbf{x}'_n) + \phi_n\right),\end{aligned}$$

see Figure 8. To take the periodicity of the phase into account, we define the total phase error at vertex 0 with respect to its neighbors $1, \ldots, N$ as

$$E_0 = \frac{1}{N}\sum_{n=1}^{N} \min_{i \in \mathbb{Z}} \left(\frac{\psi_n - \psi_0}{2\pi} - i\right)^2.$$

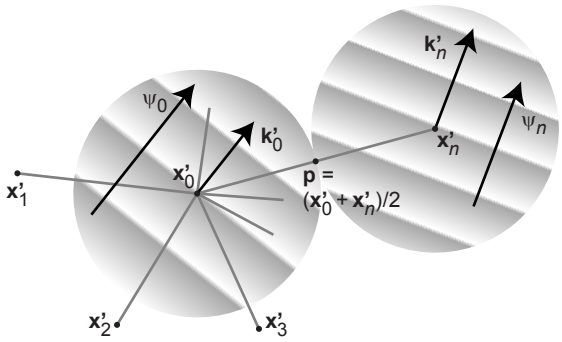

Figure 8: *The quadratic phase errors on the centers of all connecting lines to the neighbors are averaged.*

To achieve realistic results, it turned out to be sufficient to apply a gradient descent via

$$\phi_0 \leftarrow \phi_0 - \varepsilon \frac{\partial E_0}{\partial \phi_0}, \tag{6}$$

where ε is a small positive number. The descent is applied per time step, ensuring temporal coherence with no sudden leaps of wrinkles. One can choose a smaller ε to create a smoother, floating look. If ε is set too high, the process becomes instable and yields oscillating motions. Thus, to allow a fast but stable relaxation the software prototype allows to run several rounds of relaxation per frame—with correspondingly lower ε. However, for all practical purposes a single relaxation step proved to be sufficient. Furthermore, no objectionable trapping into shallow local minima could be noticed in the experiments.

If Equation 6 is applied as such, regions of wrinkles that are separated by flat parts may influence each other: Even though not visible to the user, the direction vectors in the flat part will align themselves to those of the adjacent wrinkled parts—and act back on the direction vectors there. Thus, we elected to weigh every neighbor's contribution to the error E_0 by $\max(\min(4h_n - 0.2, 1.0), 0.0)$, which smoothly attenuates the effect of neighbors with amplitudes h_n less than 0.3.

6. Final rendering

The vertex and pixel shader that are responsible for the final rendering compute the lighting and deform the texture coordinates to create the illusion of a surface that is wrinkled. For deforming the texture coordinates, a simple approximation related to parallax mapping turns out to be sufficient. The method used for lighting is more related to bump mapping than to normal mapping: It is directly based on height data and does not rely—as most GPU-based methods do—on an auxiliary texture containing the deformed normal vectors. Such a normal map would have to be updated continuously to conform to the height field

In the former steps, we have unified vertices that occupy the same position in space but have different texture coordinates to allow for texture seams. For the final rendering, the original, possibly duplicated vertices have to be used. They are equipped with an additional datum: the ID number of the corresponding vertex in the smaller, unified set.

6.1. Lighting

The vital ingredient to the lighting computation is a per-pixel normal, to be determined in post-deformation space. Assume we are given a base plane $P \subset \mathbb{R}^3$ with the (unit) normal vector $\mathbf{n}'$ and a height field $h : P \to \mathbb{R}$. The height field h shifts any point $\mathbf{x}'$ of the plane to $\mathbf{x}' + h(\mathbf{x}')\mathbf{n}'$. At such a point, the resulting deformed surface has a (non-normalized) normal of

$$\mathbf{n}' - \frac{\partial}{\partial \mathbf{x}'} h(\pi_P(\mathbf{x}')), \tag{7}$$

where π_P is the perpendicular projection onto the plane, see Figure 9. This equation can be understood as a lean variant of Blinn's original bump mapping formula [Bli78], which employed *uv* parameterization.

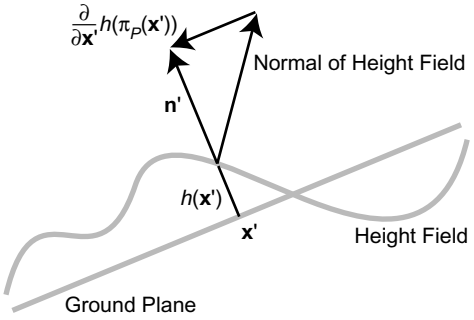

Figure 9: *The per-pixel normals can be computed with the help of the gradient of the height field.*

© The Eurographics Association and Blackwell Publishing 2006.

Returning to the curved model's surface, we can approximately identify the plane P with both the tangent plane through a vertex and with a triangular face containing this vertex. The $\mathbf{k}'$ of Equation 4 will lie in the tangent plane, so that we can immediately insert arbitrary $\mathbf{x}'$ into h according to Equation 5 and find $h(\mathbf{x}') = h(\pi_P(\mathbf{x}'))$. If we assume that the $\mathbf{k}'$ and the amplitude vary slowly on the surface we find:

$$\begin{aligned}
&\frac{\partial}{\partial \mathbf{x}'} h(\pi_P(\mathbf{x}')) \\
&\approx -\frac{2\pi}{W}\alpha \mathbf{k}'_A h_A \sin\left(\frac{2\pi}{W}\mathbf{k}'_A \cdot (\mathbf{x}' - \mathbf{x}'_A) + \phi_A\right) \\
&\quad -\frac{2\pi}{W}\beta \mathbf{k}'_B h_B \sin\left(\frac{2\pi}{W}\mathbf{k}'_B \cdot (\mathbf{x}' - \mathbf{x}'_B) + \phi_B\right) \\
&\quad -\frac{2\pi}{W}\gamma \mathbf{k}'_C h_C \sin\left(\frac{2\pi}{W}\mathbf{k}'_C \cdot (\mathbf{x}' - \mathbf{x}'_C) + \phi_C\right). \qquad (8)
\end{aligned}$$

The arguments of the cosines such as $\mathbf{k}'_A \cdot (\mathbf{x}' - \mathbf{x}'_A) + \phi_A$ vary linearly in space, so they can be computed efficiently in a vertex shader. However, the contributions of the three vertices A, B, C of every triangle have to be treated separately. This requires a clear assignment of variables to each of the three vertices, which forbids that vertices are shared between triangles. Thus, for the final rendering a list of separate triangles has to be used (see Figure 2), not the indexed list of vertices as which a mesh is typically given.

To evaluate Equation 7, only the normal $\mathbf{n}'$ remains to be specified. Since we are not dealing with a plane but with a curved surface, the obvious choice is to use Phong interpolation to determine $\mathbf{n}'$ per pixel.

A faster approach to compute the gradient of h would be to use the instructions `ddx` and `ddy` offered by HLSL. They compute differences (that is: they approximate partial derivatives) in screen space. With them, the gradient of h could be derived from the values of h. However, the partial derivatives in screen space would have to be transformed to 3D. Furthermore, these instructions operate by looking at 2×2 pixels at a time. This leads to objectionable block-like artifacts, see Figure 10.

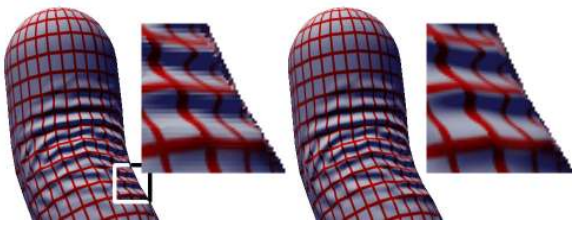

Figure 10: *HLSL's partial-derivative instructions can simplify the computation of normals, but lead to block-like artifacts (left: with partial derivatives, right: according to Equation 8, insets: fourfold magnification).*

6.2. Texturing

A gross deformation of the texture already takes place because it is attached to the vertices. What is needed in addition are small-scale parallax effects around the wrinkles. The method employed for this is an extension of parallax mapping [KTI*01] to curved surfaces with an arbitrary, non-regular texture parameterization.

Consider a surface point $\mathbf{x}'$ at which the height field has the value $h(\mathbf{x}')$, the (unit) normal is $\mathbf{n}'$, and the vector $\mathbf{v}$ (not necessarily normalized) points to the viewer, see Figure 11. If one assumes that both the curvature of the base surface and that of the height field are negligible, the viewing ray intersects the deformed surface at a position shifted by

$$\Delta\mathbf{x}' = \left(\frac{\mathbf{v}}{\mathbf{v}\cdot\mathbf{n}'} - \mathbf{n}'\right) h(\mathbf{x}'). \qquad (9)$$

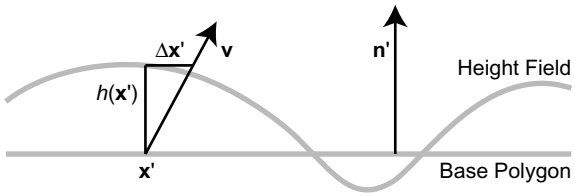

Figure 11: *The approximation for the texture parallax assumes a slowly varying height field or near-normal viewing.*

For oblique viewing, $\mathbf{v}\cdot\mathbf{n}'$ becomes small and the approximation is no longer valid, see Figure 12. To cope with this, we do not introduce a hard limit [Wel04] to the offset, but use a smooth cut-off:

$$\Delta\mathbf{x}' = \frac{(\mathbf{v} - (\mathbf{v}\cdot\mathbf{n}')\mathbf{n}')\mathbf{v}\cdot\mathbf{n}'}{(\mathbf{v}\cdot\mathbf{n})^2 + 0.1|\mathbf{v}|^2} h(\mathbf{x}'). \qquad (10)$$

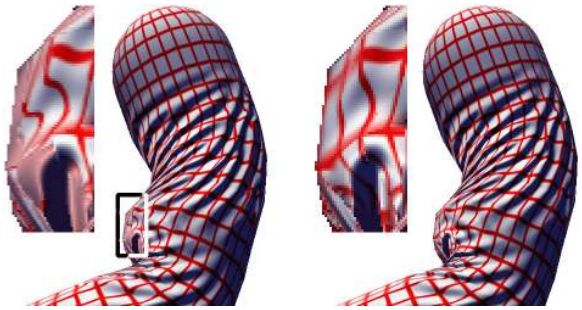

Figure 12: *The approximation by Equation 9 (left) can be adapted according to Equation 10 to also handle the horizon well (right).*

The shift $\Delta\mathbf{x}'$ in $\mathbf{x}'$ corresponds to a shift $(\Delta u, \Delta v)$ in the texture coordinates uv. Consider a triangle ABC of the mesh. Define

$$\mathbf{b}' = \mathbf{x}'_B - \mathbf{x}'_A, \qquad \mathbf{c}' = \mathbf{x}'_C - \mathbf{x}'_A.$$

© The Eurographics Association and Blackwell Publishing 2006.

Assuming that the uv are linearly interpolated over the plane in which the triangle resides, we find

$$\Delta u = \frac{(u_B - u_A)\mathbf{n}' \times \mathbf{c}' + (u_C - u_A)\mathbf{b}' \times \mathbf{n}'}{\mathbf{n}' \cdot (\mathbf{c}' \times \mathbf{b}')} \cdot \Delta \mathbf{x}', \qquad (11)$$

and similarly for Δv, where u_A, u_B, and u_C denote the values of the u coordinate at the vertices of the triangle. The fraction that appears to the left of $\Delta \mathbf{x}'$ can be computed in the vertex shader. It varies from triangle to triangle, which may introduce jumps in the deformation of the texture.

6.3. Wrinkle profiles

The cosine in Equation 3 may be replaced by another function f that is 2π-periodic and obeys $f(\phi) = f(-\phi)$ for all ϕ. The latter symmetry is necessary because the sign of the direction vectors $\mathbf{k}$ is not determined uniquely. If f was asymmetric and if $\mathbf{k}$ pointed down at one vertex and up at a neighbor, it would be impossible to align the wrinkles using the phase offset ϕ alone.

To compute the deformation of the texture, f is needed in Equation 5; to compute the lighting, its derivative $df/d\phi$ has to be known for Equation 8. We store both f and $df/d\phi$ as linearly interpolated value tables in textures. This ensures full generality, see Figure 13, and avoids possibly expensive computations of transcendental functions such as the cosine.

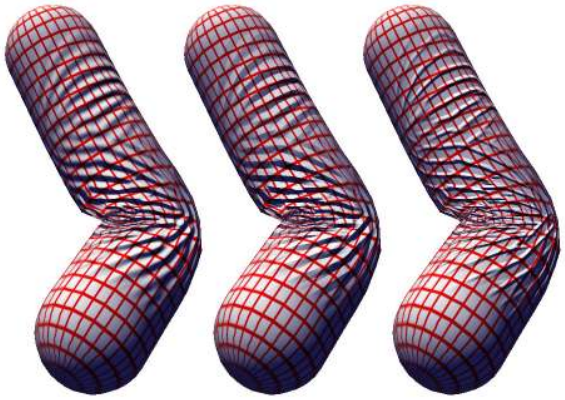

Figure 13: *By filling a texture appropriately, wrinkles with a sinusoidal profile (left) may for instance be replaced by accordion pleats (middle), or pencil pleats (right).*

All three vertices of a triangle contribute to a single pixel. Whereas the three phase values (i. e., the arguments of the cosine in Equations 5 and 8) can be determined per vertex and interpolated linearly, both f and $df/d\phi$ have to be evaluated per pixel for all of these three phase values. Thus, a 1D texture with two color channels is used as a look-up table, which stores 32 values of f and $df/d\phi$. The automatic wrapping of the texture coordinates on the GPU turns this table into a periodic function.

7. Painting rest-pose wrinkles

Typical garments show wrinkles also in the rest pose of a 3D character. Hence, the software prototype allows to paint wrinkles onto the surface or to generate them randomly. These interact seamlessly with the dynamic wrinkles.

The user can rotate the 3D object and choose the direction of the wrinkles to be painted freely with respect to xy screen space: left-right, up-down, and so on, see Figure 14. As the user drags the mouse over the screen, this direction is projected into the tangent space of the nearest vertex under it. The vector is blended into an off-screen buffer whose texels correspond to vertices of the mesh and are initially set to the null vector. Thus, the content of this buffer describes the amount and the direction of the intended rest-pose wrinkles via the length and the direction (in rest-pose space) of one tangent vector per vertex.

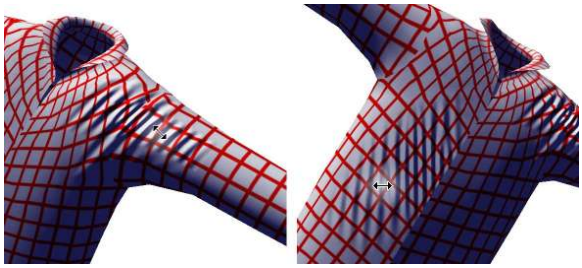

Figure 14: *The wrinkle direction specified in screen space during the painting of rest-pose wrinkles is projected into tangent space.*

The contents of this buffer are used to bias the computation of the dynamic wrinkle direction and amplitude data according to Subsection 4.2. The idea is to modify the 2×2 matrix M that describes the local linear approximation of deformation according to Equation 1, that is

$$\mathbf{v}_i' \approx M\mathbf{v}_i.$$

We can force wrinkles along some specified direction if we shorten the rest-post space vector $\mathbf{v}_i$ along the intended wrinkle direction before applying M. A simple way for doing so is to replace M by the product

$$M(E - \mathbf{q} \otimes \mathbf{q}),$$

where E denotes the 2×2 identity matrix, and $\mathbf{q}$ is the 2D tangent-space version of the vector $\mathbf{p} \in \mathbb{R}^3$ read from the bias texture filled by painting:

$$\mathbf{q} = \begin{pmatrix} \mathbf{t} \cdot \mathbf{p} \\ \mathbf{b} \cdot \mathbf{p} \end{pmatrix}$$

The painting routine ensures that the vector $\mathbf{p}$ and hence also the vector $\mathbf{q}$ always have a length smaller than one.

© The Eurographics Association and Blackwell Publishing 2006.

8. Results

An animation with known ground truth serves as test of the geometric computations: A flat square composed of 148 triangles is contracted to half its original width. Even though the network of the triangles is strongly irregular, ideally a pattern of parallel wrinkles should appear, see Figure 15. As the results show, the relaxation process helps greatly. The global minimum—that is: parallel folds with no bifurcations—is not attained completely, however. But actually this makes the result *more* plausible. Since the relaxation changes the wave phase at the vertices only gradually, the height maxima and minima tend to stay bound to the deforming geometry, which adds to the plausible behavior. With accordion-type wrinkles one observes the zigzag profile that should emerge from a compression by 50 percent.

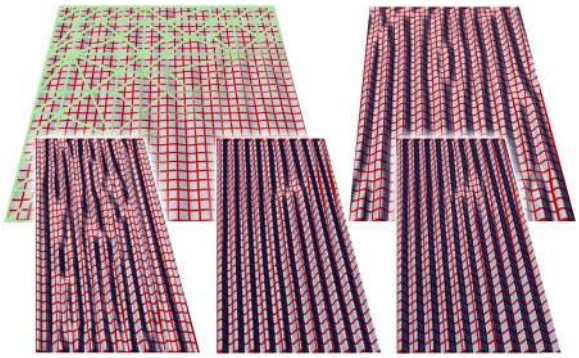

Figure 15: *An irregularly tessellated surface is subjected to compression (upper left:* 98 *%, upper right:* 75 *%, lower row:* 50 *%). Without relaxation (lower left), strong irregular wrinkles appear. The accordion pleating (lower right) displays the right amount of folding.*

The extraction of the deformation data and the lighting work well even for large wrinkles. However, the simplified parallax computation of Equation 10 and the linear per-triangle approximation according to Equation 11 become obvious for large wrinkles, see Figure 16. This is exacerbated by the growth of the wrinkles' amplitude with the wavelength, see Equation 2. The lack of occlusion effects and deformation of the silhouette is of much less importance.

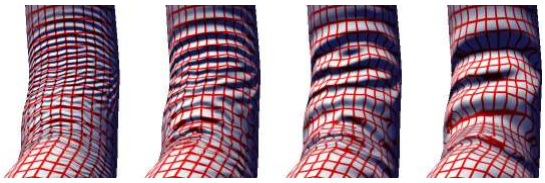

Figure 16: *The computation of the wrinkle field and the lighting is uncritical even for very large folds. However, they lead to visible artifacts in the texture deformation.*

Speed benchmarks were done on an Intel® Pentium® 4 running at 2.5 GHz with an Nvidia® GeForce® 6800 GT graphics card displaying 1280×1024 pixels full screen with no vertical synchronization and no back face culling. Table 1 shows the timing results for three typical models, which are also depicted in the figures of this paper.

Name	# Vertices	# Pixels (average)	fps
Shirt	455	≈ 330.000	328
Zeppelin	508	≈ 260.000	540
Curtain	92	≈ 505.000	537

Table 1: *The artificial details produced by wrinkling allow using coarse models to achieve highly interactive speeds.*

The contribution of the different shaders to the computational load is shown in Table 2. To explore the possible range, model "A" is coarse but large on screen (100 vertices, approx. 940.000 pixels); model "B" is highly detailed with a small size on screen (5977 vertices, approx. 55.000 pixels). The contribution of each of the four steps is deduced from the change in the frame rate when one of the steps is skipped. Clearly, the final rendering is the most time-intensive step, partially due to the high number of texture reads.

Stage	# Shader instructions		Contribution	
	Vertex	Pixel	A	B
Skin	$13M+20$	2	0.06 ms	0.25 ms
Crush	8	$28N+102$	0.08 ms	1.83 ms
Relax	7	$33N+23$	0.11 ms	1.75 ms
Render	67	47	3.11 ms	5.01 ms
Total time incl. non-shader part			3.45 ms	8.90 ms

Table 2: *The relative workload of the shaders depends on the choice of the model (for "A" and "B" see text). The maximum number of bone influences per vertex is M; the number of neighbors of a vertex is N.*

9. Conclusion and outlook

We have presented a shader-based solution to create folds and wrinkles in real time. The wavelength, the profile, and rest-pose wrinkles can be controlled easily. As the benchmarks demonstrate, the method can be employed in speed-sensitive applications such as games. The speed can be increased further by executing not all of the four steps for every frame. In many situations it may suffice to run the "Crush" and the "Relax" steps only every second frame or even more rarely.

To further improve the plausibility of the results, one can control the wavelength of the wrinkles through another texture. Whereas the folds look plausible for cloth or for plastic foil, skin simulations would profit from stronger irregularities. These can be achieved by skipping the relaxation or by creating disturbances through a user-defined texture.

© The Eurographics Association and Blackwell Publishing 2006.

Future work can combine the described technique for wrinkles with a physical simulation or a morph animation for the coarse mesh. On top of that, improved methods for parallax effects according to Section 2 may address occlusion and self shadowing, at corresponding costs in terms of rendering speed. Three of the four processing steps will profit greatly from the geometry shader units in next-generation graphics hardware [Mic05], which allow direct access to both a vertex' neighbors and all vertices of a triangle.

References

[BKN02] BANDO Y., KURATATE T., NISHITA T.: A simple method for modeling wrinkles on human skin. In *Proceedings of PG '02: Pacific Conference on Computer Graphics and Applications* (2002), pp. 166–175.

[Bli78] BLINN J. F.: Simulation of wrinkled surfaces. In *SIGGRAPH '78: Proceedings of the 5th Annual Conference on Computer Graphics and Interactive Techniques* (1978), pp. 286–292.

[CGW*05] CUTLER L. D., GERSHBEIN R., WANG X. C., CURTIS C., MAIGRET E., PRASSO L., FARSON P.: An art-directed wrinkle system for CG character clothing. In *Proceedings of SCA '05: ACM SIGGRAPH/Eurographics Symposium on Computer Animation* (2005), pp. 117–125.

[CMT05] CORDIER F., MAGNENAT-THALMANN N.: A data-driven approach for real-time clothes simulation. *Computer Graphics Forum 24*, 2 (2005), 173–183.

[CN02] COMBAZ J., NEYRET F.: Painting folds using expansion textures. In *Proceedings of PG '02: Pacific Conference on Computer Graphics and Applications* (2002), pp. 176–183.

[DJW*06] DECAUDIN P., JULIUS D., WITHER J., BOISSIEUX L., SHEFFER A., CANI M.-P.: Virtual garments: a fully geometric approach to cloth design. *Computer Graphics Forum 25*, 3 (2006), this issue.

[HBVMT99] HADAP S., BANGERTER E., VOLINO P., MAGNENAT-THALMANN N.: Animating wrinkles on clothes. In *Proceedings of VIS '99: The Conference on Visualization* (1999), pp. 175–182.

[Her01] HERMAN D. L.: Using precomputed cloth simulations for interactive applications. In *SIGGRAPH '01 Sketches & Applications* (2001), p. 273.

[KC02] KANG Y.-M., CHO H.-G.: Bilayered approximate integration for rapid and plausible animation of virtual cloth with realistic wrinkles. In *Proceedings of Computer Animation 2002* (2002), pp. 203–214.

[KCCL01] KANG Y.-M., CHOI J.-H., CHO1 H.-G., LEE D.-H.: An efficient animation of wrinkled cloth with approximate implicit integration. *The Visual Computer 17* (2001), 147–157.

[KG90] KUNII T. L., GOTODA H.: Singularity theoretical modeling and animation of garment wrinkle formation processes. *The Visual Computer 6* (1990), 326–336.

[KG03] KONO H., GENDA E.: Wrinkle generation model for 3D facial expression. In *SIGGRAPH '03 Sketches & Applications* (2003).

[KTI*01] KANKEKO T., TAKAHEI T., INAMI M., KAWAKAMI N., YANAGIDA Y., MAEDA T., TACHI S.: Detailed shape representation with parallax mapping. In *Proceedings of ICAT 2001: International Conference on Artificial Reality and Telepresence* (2001), pp. 205–208.

[LC04] LARBOULETTE C., CANI M.-P.: Real-time dynamic wrinkles. In *Proceedings of CGI '04: Computer Graphics International* (2004), pp. 522–525.

[Mic05] MICROSOFT CORP.: DirectX 10 SDK. `http://msdn.microsoft.com/directx/`, 2005. December 2005 Update.

[MTV05] MAGNENAT-THALMANN N., VOLINO P.: From early draping to haute couture models: 20 years of research. *The Visual Computer 21* (2005), 506–519.

[New05] NEWTEK CORP.: LightWave 3D 9.0. `http://www.newtek.com/`, 2005.

[NG95] NG H., GRIMSDALE R. L.: GEOFF—a geometrical editor for fold formation. In *Image Analysis Applications and Computer Graphics*, R. Chin et al., (Ed.), vol. 1024 of *Lecture Notes in Computer Science*. Springer, 1995, pp. 124–131.

[OM01] OSHITA M., MAKINOUCHI A.: Real-time cloth simulation with sparse particles and curved faces. In *Proceedings of Computer Animation 2001* (2001), pp. 220–227.

[POC05] POLICARPO F., OLIVEIRA M. M., COMBA J. L. D.: Real-time relief mapping on arbitrary polygonal surfaces. In *Proceedings of SI3D '05: Symposium on Interactive 3D Graphics and Games* (2005), pp. 155–162.

[Tat05] TATARCHUK N.: Practical dynamic parallax occlusion mapping. In *SIGGRAPH 2005 Sketches* (2005).

[Wel04] WELSH T.: Parallax mapping with offset limiting: A per-pixel approximation of uneven surfaces. White Paper, `http://www.infiscape.com/rd.html`, 2004. Rev 0.4, January 2004.

[WWT*03] WANG L., WANG X., TONG X., LIN S., HU S., GUO B., SHUM H.-Y.: View-dependent displacement mapping. *ACM Trans. Graph. 22*, 3 (2003), 334–339. (Proceedings of SIGGRAPH 2003).

[WWY06] WANG Y., WANG C. C. L., YUEN M. M. F.: Fast energy-based surface wrinkle modeling. *Computers&Graphics 30* (2006), 111–125.

© The Eurographics Association and Blackwell Publishing 2006.