High-Performance Graphics 2019
T. Foley and M. Steinberger
(Guest Editors)
Author's Preprint

Volume 38 (2019), Number 8

Distortion-Free Displacement Mapping

Tobias Zirr[1] Tobias Ritschel[2]

[1] Karlsruhe Institute of Technology [2] University College London

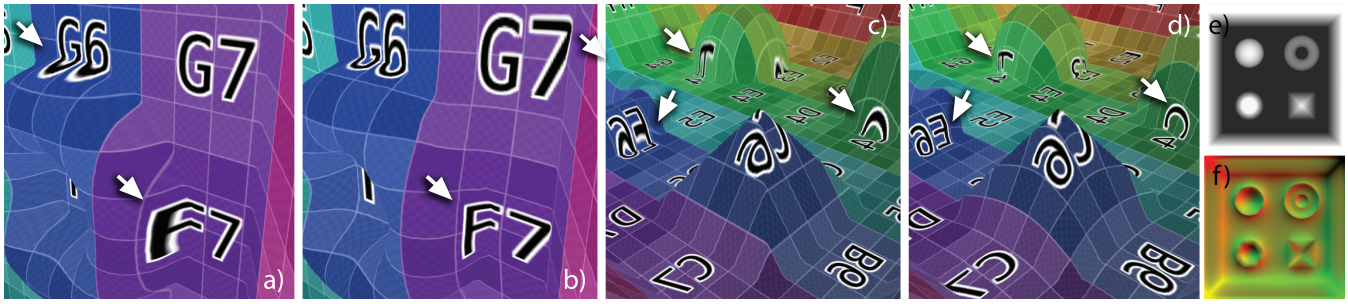

Figure 1: *Displacement mapping a textured surface introduces distortions as can be seen for the stretched letters (arrows in* (a) *and* (c)*). Our approach corrects this (arrows in* (b) *and* (d)*) by counter-distorting the texture map* (f) *according to the displacement map* (e)*.*

Abstract

Displacement mapping is routinely used to add geometric details in a fast and easy-to-control way, both in offline rendering as well as recently in interactive applications such as games. However, it went largely unnoticed (with the exception of McGuire and Whitson [MW08]) that, when applying displacement mapping to a surface with a low-distortion parametrization, this parametrization is distorted as the geometry was changed by the displacement mapping. Typical resulting artifacts are "rubber band"-like distortion patterns in areas of strong displacement change where a small isotropic area in texture space is mapped to a large anisotropic area in world space. We describe a fast, fully GPU-based two-step procedure to resolve this problem. First, a correction deformation is computed from the displacement map. Second, two variants to apply this correction when computing displacement mapping are proposed. The first variant is backward-compatible and can resolve the artifact in any rendering pipeline without modifying it and without requiring additional computation at render time, but only works for bijective parametrizations. The second variant works for more general parametrizations, but requires to modify the rendering code and incurs a very small computational overhead.

CCS Concepts

• ***Computing methodologies*** → ***Texturing;***

1. Introduction

Displacement mapping is a common and computationally efficient way to add details to a shape which would be inefficient or impossible to store as a 3D surface. It is therefore popular both in offline rendering for film production and in computer games. Also, artists prefer to directly manipulate a height field in a painting program such as Autodesk Mudbox or ZBrush over manipulation of individual polygonal primitives. Ideally, a displacement-mapped surface should have the same visual appearance as a full polygonal mesh. While this has become reality in the industry in terms of the surface geometry, it is not true in terms of texturing. Whereas full polygonal meshes are now routinely parametrized to achieve low-distortion texture mapping, this is not possible for displacement-mapped surfaces, a shortcoming addressed in this paper (Fig. 1).

Besides a discussion in McGuire and Whitson [MW08], the artifact in question went surprisingly unnoticed. The artifact has the following cause: While a base surface has a low-distortion parametrization, displacement mapping deforms the "base" surface but the parametrization of the base mesh remains unaffected. Consequently the parametrization becomes distorted: e. g., circles in texture space that were circles in world space are not circles in world space anymore. This cannot be fixed by simply parametrizing the result of the displacement mapping, as representing this surface ideally is "avoided" , e. g., in the REYES architecture [CCC87]

because it would not fit into memory, or because it could not be handled by common parametrization methods that solve sparse linear system with a size in the order of the number of vertices [DMK03, FH05]. At the same time, the artifact is visually important because the mammalian (i. e., human) visual system perceives textures as anisotropic frequency content which is substantially different [HW62]. Even worse, the systematic error is correlated with geometry, resulting in an additional but wrong perceptual texture-density cue [BL76].

As a solution, we devised a special correction parametrization that changes the texture mapping process, such that when it is later used for a specific displacement map, the distortion is minimized.

2. Background

Displacement mapping was introduced by Cook in 1984 [Coo84]. It is nowadays routinely used in interactive graphics, thanks to efficient GPU implementations [SKU08]. The two dominant methods on GPUs are either to tesselate the coarse base geometry [Tat07] or to use ray casting in a height field enclosed by a bounding volume [PHL91]. Our work is orthogonal to how the displacement is produced. Generalization to vectorial displacements are possible [WTL*04] and supported by our approach.

Displacement mapping is closely related to the idea of storing the entire geometry in a regular structure such as in geometry images [GGH02] and multi-chart geometry images [SWG*03]. These works map arbitrary genus-0 surfaces to the two-dimensional disk, which is similar to the problem we face. The idea to resample a signal, such that certain problems are avoided, was proposed for surface attributes by Sander et al. [SGSH02]. We identify the specific issue in the relation of parametrization and displacement mapping and find a simple solution tailored to displacement mapping that could be combined with signal-specialized parametrization but is not identical to it as the signal and the change of parametrization are in a more general relation. We also show how re-sampling is not the best choice in terms of quality for our problem in all situations, e. g., if distortions are extreme or there is no bijective mapping between the displacement map and the texture. In both cases, a simple shader modification allows for better results.

The general topic of parametrization is covered in a survey [FH05] and a recent course [HPS08], in particular page 41 and following. General parametrization, in particular when multiple charts are involved, can be computationally demanding: a 4M vertices (equivalent to a 2000×2000 displacement map) mesh of the pelvis takes in the order of hundreds of seconds for an already well-optimized state-of-the-art parametrization method (ABF++) [SLMB05, Fig. 9 and Table III]. We will show how the problem of correcting displacement map parametrization has a more practical solution by applying the adapted MIPS (Mostly Isometric Parameterization) energy [DMK03] to displacement maps.

McGuire and Whitson [MW08] have introduced an indirection map, mapping texture coordinates that were distorted by displacement mapping to undistorting texture coordinates, thus obtaining a more uniform resolution across the entire displaced mesh. For their undistortion, they implement a CPU-based quasi-conformal multi-scale spring relaxation algorithm that first re-distributes the subdivisions of the displaced mesh, and then computes the inverse mapping by rasterization. In contrast, we directly compute the correction map, without additional resampling, using a general energy-based approach with massive parallelization on the GPU (GLSL code is included in the appendix). We show results for a specific energy function for good area- and angle preservation that has been shown useful in previous parametrization work [DMK03], but other proven energies are supported. In addition, our approach allows adapting boundary constraints optimized for the use case, e. g., tiled or non-tiled textures.

3. Our approach

We will start the exposition of our approach by an overview in the next section Sec. 3.1 before presenting its two main steps in detail in Sec. 3.2 and Sec. 3.3.

3.1. Overview

Our approach consists of two main steps: First, computing a correction field and second applying this correction.

Input to the first step (Sec. 3.2) is the displacement map. Output of this step is a *correction map*, a two-dimensional field of two-dimensional vectors c that, when applied to a texture coordinate of the polygonal surface, will correct the distortion caused by displacement mapping. Fig. 2 shows an example for the one-dimensional case. The texture is assumed to have been stationarized [MJH*17], i. e., that its statistics are spatially invariant. A typical example is wood, stone or fabric. Textures correlated with the underlying displacement map, e.g. containing distinct textured parts of complex objects, can be less suited for our approach. For more basic correlations, such as the darkening in the creases of cobblestones, our approach easily allows the addition of additional fixation constraints (see Sec. 3.2.2).

The second step (Sec. 3.3) uses the correction field to improve sampling of other, e. g., diffuse, textures. This improvement is independent of the actual displacement mapping technique used

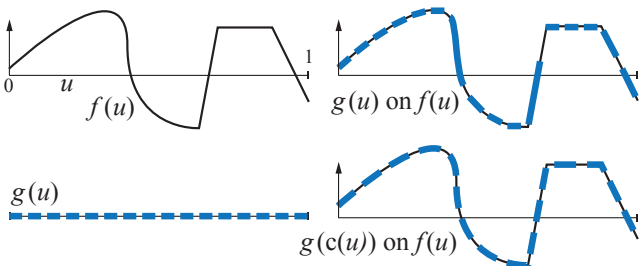

Figure 2: *Displacement mapping texture distortion in flatland: The displacement function f maps the real unit interval $\mathcal{U}$ to a point in $\mathbb{R}^2$. The texture function g is also defined on $\mathcal{U}$ and in this example produces blue dots. Common texturing makes equally-sized steps in u, resulting in a distortion in the 2D target space (longer and shorter blue boxes). Our approach finds a correction c, such that using $g(c(u))$ has no distortion (boxes of same length).*

[SKU08]. In the first variant (Sec. 3.3.1), the undistortion field computed is stored into an additional map. When any texture is to be read at coord $\mathbf{u}$, it is read at the coord $c(\mathbf{u})$ instead. This delivers good undistortion quality and works with texture maps in arbitrary alignment relative to the displacement map, but requires to change the shader. In the second variant (Sec. 3.3.2), the texture maps are directly resampled. This results in a fast, simple and backward-compatible solution, as the shader code does not need to be changed. However, the resulting quality is lower due to resampling and it does not work with unaligned texture maps.

3.2. Computing the undistortion

Input Displacement maps can either come as height field displacement maps or as vector displacement maps. A height field can be converted into a vector displacement map by scaling the normal at each point by the height. The vector displacement map can be converted into a parametric 3D surface by adding the vector displacement to a parametric base surface. In order to unify the handling and for ease of notation, our derivation directly uses a parametric *displaced surface* $f(\mathbf{u}) \in \mathcal{U} \to \mathbb{R}^3$ defined on a chart $\mathcal{U} \subset (0,1)^2 \subseteq \mathbb{R}^2$ which is homeomorphic to a disk and has a boundary $\mathcal{B} \subseteq \mathcal{U}$. Depending on the type of displacement used (height field, vectorial), conversions need to be done when evaluating f.

Output As output, we would like to compute a function $c(\mathbf{u}) \in \mathcal{U} \to \mathcal{U}$ that remaps texture coordinates $\mathbf{u}$ to new texture coordinates $c(\mathbf{u})$ such that the magnitude of differential changes $\partial_{\mathbf{u}} c(u)$ with respect to $\mathbf{u}$ becomes proportional to the magnitude of differential changes $\partial_{\mathbf{u}} f(u)$ in world space.

Please note that c is never used to change evaluation of the displacement maps, but only to warp the texture coordinates of one or multiple texture maps that are subsequently applied to the displaced surface, such as diffuse, specular, or normal maps. Without loss of generality, such maps are henceforth denoted as $g(\mathbf{u}) \in \mathcal{U} \to \mathbb{R}^n$, since their nature does not have an influence on the computation of the undistortion.

3.2.1. Distortion Energy

The first fundamental form $\mathcal{I}f$ of a parametric map f defined as

$$\mathcal{I}f = (\nabla_u f)^T \nabla_u f, \tag{1}$$

describes how a mapping from 2D to 3D distorts angles and area [DMK03]. In order to minimize distortion, we want to preserve both angles, found in the ratio of smallest and largest eigenvalue, and area, found in the determinant of $\mathcal{I}$. An angle- and area-preserving mapping is called *isometric* and its fundamental form is the identity. Such mappings exist only for a very limited class of surfaces in practice, so a tradeoff between angle and area preservation has to be made. Therefore, we want to find a function c, such that $\mathcal{I}f(c^{-1}(\mathbf{u})) =: \mathcal{I}_{f/c}$ is as close as possible to the identity. Area distortion can be measured by considering an infinitely small quad mapped to a trapezoid with area $\sqrt{\det \mathcal{I}_f}$. If $x = 1$, area is preserved. Similar, angle preservation corresponds to the ratio of the larger and the smaller eigenvalue $\lambda_{\max}/\lambda_{\min}$ of f. Here, $x = 1$ preserves angle as well.

Degener et al. [DMK03] propose to find the mapping c using the function $x + \frac{1}{x}$ to penalize deviations from isometry. Distortion occurs at $x > 1$. They minimize the energy

$$E(c,\mathbf{u}) = \left(\sqrt{\det \mathcal{I}_{f/c}(\mathbf{u})} + \frac{1}{\sqrt{\det \mathcal{I}_{f/c}(\mathbf{u})}}\right)^{\theta} \left(\sqrt{\frac{\lambda_{min}(\mathcal{I}_{f/c}(\mathbf{u}))}{\lambda_{max}(\mathcal{I}_{f/c}(\mathbf{u}))}} + \sqrt{\frac{\lambda_{max}(\mathcal{I}_{f/c}(\mathbf{u}))}{\lambda_{min}(\mathcal{I}_{f/c}(\mathbf{u}))}}\right) \tag{2}$$

with respect to the boundary condition that $c(\mathbf{u}) = \mathbf{u}$ if $u \in \mathcal{B}$.

3.2.2. Optional Correlation Constraints

Correlations of color maps with displacement maps, e.g. specific materials in creases or important straight lines, can be simply enforced by the addition of penalty terms for the movement of specific texels. We introduce an optional fixation factor $\gamma(\mathbf{u}) \in [0,1]$ that defaults to 0, but lets certain coordinates $\mathbf{u}$ be fixed such that $\|c(\mathbf{u}) - \mathbf{u}\| \leq 1$ texel for $\gamma(\mathbf{u}) = 1$. The penalty term added to the energy $E(c,\mathbf{u})$ and derived by the solver is then:

$$E_\gamma(c,\mathbf{u}) := \frac{1}{2} \frac{1}{\max\left(1 - \|\gamma \mathbf{R}\,(c(\mathbf{u}) - \mathbf{u})\|^2, 0\right)},$$

where $\mathbf{R}$ is a vector of the number of texels for each dimension of $\mathbf{u}$.

3.2.3. Solver

We propose to exploit the special structure of our problem in a simple, iterative and fine-grained parallel – therefore GPU-friendly – solver that exploits the implicit 2D problem structure. The displacement map texels are interpreted as vertices in a regular grid of triangulated quads. Derivatives of the energy with respect to texture coordinates are computed for the six triangles adjacent to each texel and a local gradient descent is made. In practice c and f are discretized into images of n texels, stacked into two vectors $\mathbf{c} \in \mathbb{R}^{2\times n}$ and $\mathbf{f} \in \mathbb{R}^{3\times n}$. Initially, the correction is set to identity. Then, in each step, for texel i, the gradient of the energy ∇e in respect to the correction is computed. In the following, We will derive a closed-form expression for this gradient.

The gradient is computed in respect to a six-neighborhood $\mathcal{N}_i$ of texels adjacent to the triangles formed with texel i. Let $j = 1,\ldots,6$ be an index in this neighborhood, we write $\Delta c_{i,j} = \mathbf{c}_i - \mathbf{c}_{\mathcal{N}_{i,j}}$, $\Delta f_{i,j} = \mathbf{f}_i - \mathbf{f}_{\mathcal{N}_{i,j}}$. For triangle j adjacent to neighbour texels j and $j+1$ (modulo 6), the areas are $A^c_{i,j} = \frac{1}{2}||\Delta c_{i,j} \times \Delta c_{i,j+1}||$ and $A^f_{i,j} = \frac{1}{2}||\Delta f_{i,j} \times \Delta f_{i,j+1}||$. Then

$$E_{i,j} = \frac{1}{A^c_{i,j} A^f_{i,j}} \Big(||\Delta c_{i,j}||^2 \left\langle \Delta f_{i,j+1} - \Delta f_{i,j}, \Delta f_{i,j+1} \right\rangle + ||\Delta c_{i,j+1}||^2 \left\langle \Delta f_{i,j}, \Delta f_{i,j} - \Delta f_{i,j+1} \right\rangle + ||\Delta c_{i,j} - \Delta c_{i,j+1}||^2 \left\langle \Delta f_{i,j}, \Delta f_{i,j+1} \right\rangle \Big) \left(\frac{A^c_{i,j}}{A^f_{i,j}} + \frac{A^f_{i,j}}{A^c_{i,j}} \right)^{\theta}$$

is the energy of adjacent triangle j [HPS08]. We use $\theta = 3$ as a tradeoff between angle and area preservation.

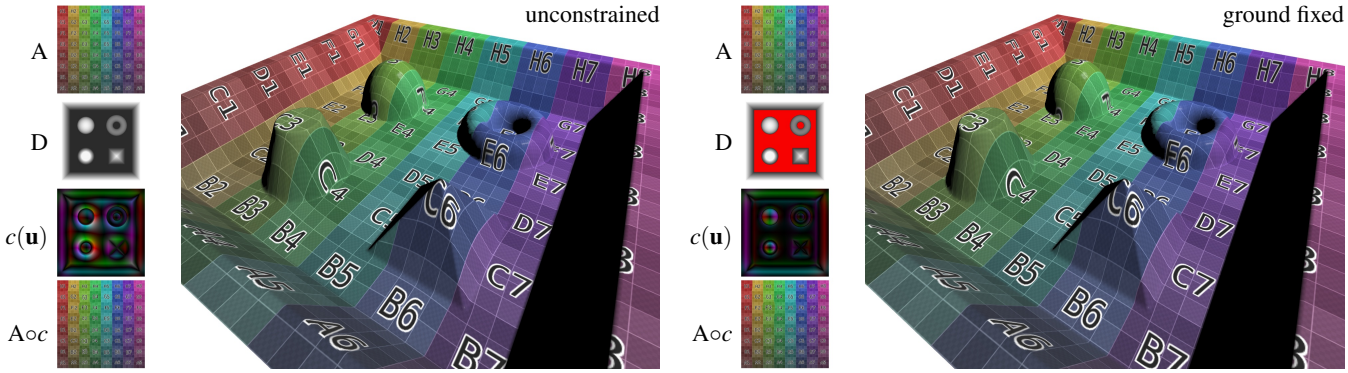

Figure 3: *Without additional constraints, our method can perturb texture maps (*A*) in arbitrary ways, potentially causing straight lines to be curved and correlations with geometry (defined by displacements* D*) to be disrespected. By adding an additional penalty term to the energy in our solver, parts of the texture maps may be fixed to certain locations. The influence of the penalty term is controlled by additional inputs to the optimizer, e.g. a certain height interval in the displacement map, or an additional artist-provided input texture. Fixed areas are overlain in red on top of the displacement map* D *on the right.*

The energy of texel i is then $E_i = \sum_{j=1}^{6} E_{i,j}$. Writing $E_{i,j} = E_{i,j}^{(1)}(E_{i,j}^{(2)} + E_{i,j}^{(3)})^{\theta}$, the gradient is

$$\begin{aligned}\nabla_c E_{i,j} =& (\nabla_c E_{i,j}^{(1)})(E_{i,j}^{(2)} + E_{i,j}^{(3)})^{\theta} \\ &+ E_{i,j}^{(1)}(\theta(E_{i,j}^{(2)} + E_{i,j}^{(3)})^{\theta-1}(\nabla_c E_{i,j}^{(2)} + \nabla_c E_{i,j}^{(3)})).\end{aligned}$$

Note that the partials of $\Delta c_{i,j}$ with respect to c are 1. Then,

$$\begin{aligned}\nabla_c E_{i,j}^{(1)} =& -\frac{E_{i,j}^{(1)}}{A_{i,j}^{c}} \nabla_c A_{i,j}^{c} \\ &+ \frac{1}{A_{i,j}^{c} A_{i,j}^{f}} \Big(\Delta c_{i,j} < \Delta f_{i,j+1} - \Delta f_{i,j}, \Delta f_{i,j+1} > \\ &\qquad + \Delta c_{i,j+1} < \Delta f_{i,j}, \Delta f_{i,j} - \Delta f_{i,j+1} > \Big), \\ \nabla_c E_{i,j}^{(2)} =& \frac{1}{A_{i,j}^{f}} \nabla_c A_{i,j}^{c}, \\ \nabla_c E_{i,j}^{(3)} =& -\frac{A_{i,j}^{f}}{(A_{i,j}^{c})^2} \nabla_c A_{i,j}^{c}, \quad \text{and} \\ \nabla_c A_{i,j}^{c} =& \begin{pmatrix} {\Delta c_{i,j+1}}^{(t)} - {\Delta c_{i,j}}^{(t)} \\ {\Delta c_{i,j}}^{(s)} - {\Delta c_{i,j+1}}^{(s)} \end{pmatrix}.\end{aligned}$$

Finally, the derivative of the energy at texel i is then $\nabla \mathbf{e}_i = \sum_{j=1}^{6} \nabla_c E_{i,j}$. With the gradient $\nabla \mathbf{e}$ for every texel at hand, we can decrease the local energy E_i of texel i by moving $\mathbf{c}_i$ along $-\nabla \mathbf{e}_i$. Since the energy is locally convex (i. e., with respect to $\mathbf{c}_i$), a step size λ_i is easily computed using a ternary search to find the minimum of E_i along the gradient line (code in the Appendix).

The gradient of the optional constraint energy $E_{\gamma,i}$ is:

$$\nabla E_{\gamma,i} = \frac{\gamma_i^2 \mathbf{R}^2 (\mathbf{c}_i - \mathbf{u}_i)}{\max\left(1 - \|\gamma_i \mathbf{R} (\mathbf{c}_i - \mathbf{u}_i)\|^2, 0\right)^2},$$

where $\mathbf{u}_i$ is the coordinate that the correction $\mathbf{c}_i$ is supposed to be pinned to for $\gamma_i = 1$.

Update It is now tempting to simply update the correction iteratively using $\mathbf{c}^{(k+1)} \leftarrow \mathbf{c}^{(k)} - \lambda^{(k)} \nabla \mathbf{e}^{(k)}$, where $\lambda^{(k)} \in \mathbb{R}^n$ is a vector that is 0 for texels on the boundary $\mathcal{B}$ or outside the chart $\mathcal{U}$ and the computed step size λ_i, otherwise. However, care has to be taken since the local steps $-\lambda_i \nabla \mathbf{e}_i$ only guarantee to decrease energy with respect to a constant neighborhood. Luckily, our problem has grid topology, so the neighborhood can be kept constant by always leaving every second row and every second column constant and changing constant rows and columns in every iteration. If $\lambda^{(k)}$ is stored in quad order (a list of quads), this corresponds to setting

$$\lambda_i^{(k)} = \begin{cases} \lambda_i, & \text{if } (i \bmod 4) = (k \bmod 4) \wedge \mathcal{U} \ni i \notin \mathcal{B} \\ 0, & \text{otherwise.} \end{cases}$$

Thus, local and global energies can only decrease and convergence is guaranteed. In essence, this is a massively parallel version of the Gauss-Seidel-like approach described in [Hor01]. While theoretically, convergence to only local minima could be possible, it is unclear whether local minima exist at all [Hor01] and in practice, we have found our parallel optimization to be reliable and fast.

Repeated Displacement Maps For repeated displacement maps (e. g., sampled with wrap mode `REPEAT` in OpenGL), improved corrections can be obtained by freeing the boundary ($\mathcal{B} = \emptyset$) and instead also applying the wrapping rules during optimization: In this case, when enumerating the neighborhood $\mathcal{N}_i$ of texel i, the texel left to the left-most texel is simply the right-most texel, etc.

Multiple Charts To optimize across multichart boundaries, we have to encode additional neighborhood information into a secondary texture [RBM06, GP09] that can then be used when the neighborhood is fetched for texels on chart boundaries. Afterwards, it might be necessary to move a small subset of boundary texels across chart boundaries in the texture maps to be undistorted, akin to filtering guard bands.

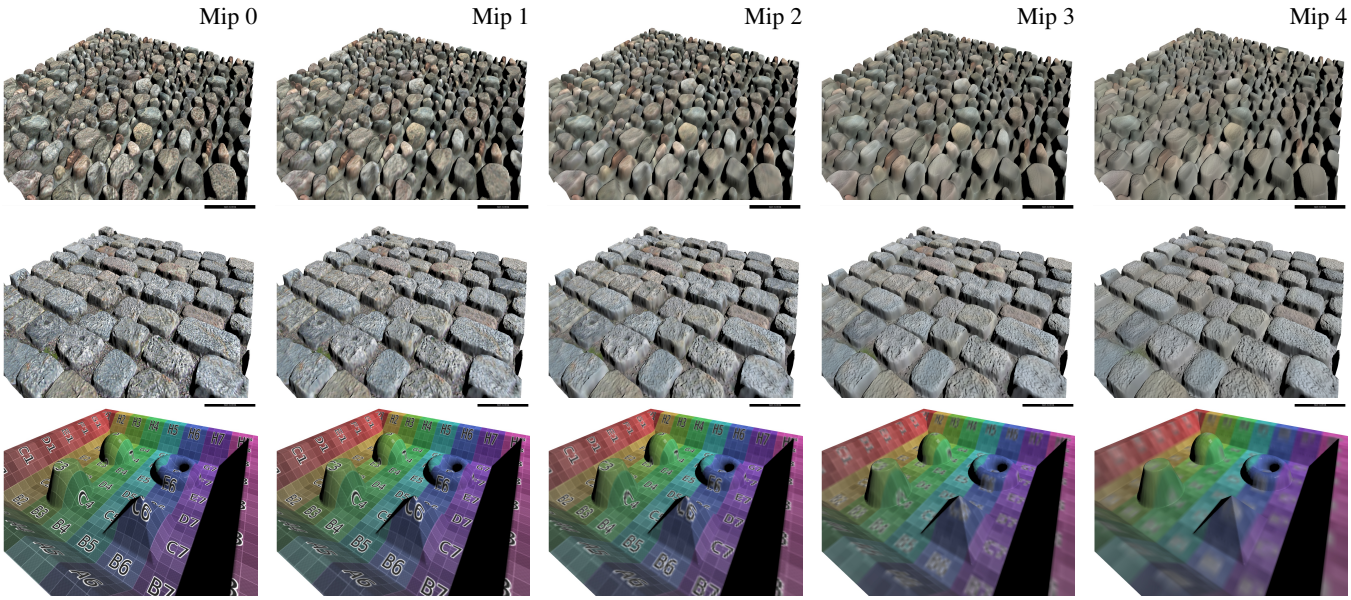

Figure 4: *Different mip levels rendered with our correction method. Mip mapping can be used for the correction texture just like for the color texture. While the correction offsets can change and lose precision in the lower-resolution mip levels, the parallel loss of accuracy in the color mip levels nicely covers up these changes. Our supplemental WebGL demo allows you to see the effects of mip mapping in motion.*

3.2.4. Implementation

We implemented the optimization in a straight-forward GLSL shader that parallelizes over texels i. e., by running exactly as many threads as there are texels and synchronizing after each iteration. All data is stored in floating-point textures (RGB for vector displacement; luminance/R for height fields).

Note that rather than storing the absolute corrected texture coordinates $\mathbf{c}$, we store relative correction offsets $\Delta\mathbf{c} = \mathbf{c} - \mathbf{u}$, which has several benefits: It naturally works with several texture wrapping modes and is easy to compress (Sec. 3.3.1).

Also note that in applications, displacement maps are often evaluated on texture boundaries rather than texel centers. In this case, computing and storing the correction map at half a texel offset can avoid unnecessary resampling artifacts. Naturally, this offset then needs to be countered when applying the correction map.

3.3. Using the undistortion

The undistortion map can be used in two ways: without resampling (Sec. 3.3.1) and with resampling (Sec. 3.3.2).

3.3.1. Without resampling

It this variant c is used in the way it is defined and computed: as a re-mapping of the coordinates. To this end, the shader using a texture map g, such as a diffuse texture, has to use an indirection when reading g: Whenever the value $g(\mathbf{u})$ is meant to be read, $g(\mathbf{u}+\Delta c(\mathbf{u}))$ is to be read instead. Here, the relative correction offset $\Delta c(\mathbf{u})$ can be sampled just like a regular texture, using interpolation and even wrapping modes. Thus, the solution is robust for several important combinations of displacement and texture maps: Both of them are allowed to wrap in an arbitrary mode such as `REPEAT`, `CLAMP`, `BORDER` while neither needs to be aligned to the other in any way. (Due to making Δc a relative (instead of absolute) correction field, it can be interpolated across tiling borders and still yields correct results.)

This approach has previously also been discussed by McGuire and Whitson [MW08], where additional reasoning about structured and unstructured texture content is provided.

Compression Before usage in rendering, the correction Δc is converted from a float into an 8-bit RG texture, to be read with bilinear filtering which naturally provides a piecewise linear parametrization, analogous to triangles. A reduction over Δc yields the smallest enclosing interval of correction offsets (typically $\sim[-0.02, 0.02]$). The range is stored and the offsets are quantized by remapping the interval to the $[0, 255]$ range. Thus, the storage overhead is kept low. Our experiments indidcate that correction maps may often be stored at half the resolution of the displacement map.

3.3.2. With resampling

In this variant, the correction field c is simply used to resample every texture g. Similar re-samplings have been used to provide higher accuracy in texture areas with more details [SGSH02]. The appealing property of this option is, that it can achieve the correction without knowing or having access to the displacement shader code and that it comes without additional computation cost at runtime. Also, it does not incur any memory overhead.

However, the usual caveats of resampling apply: the more non-uniform f is, the more sampling loss occurs. Also, a unique mapping between all locations g and f must exist and be known beforehand. This is violated if f and g differ by a transformation or if g is used in combination with different displacement maps f_1 and f_2. In such cases, g needs to be aligned to f by separate resampling steps.

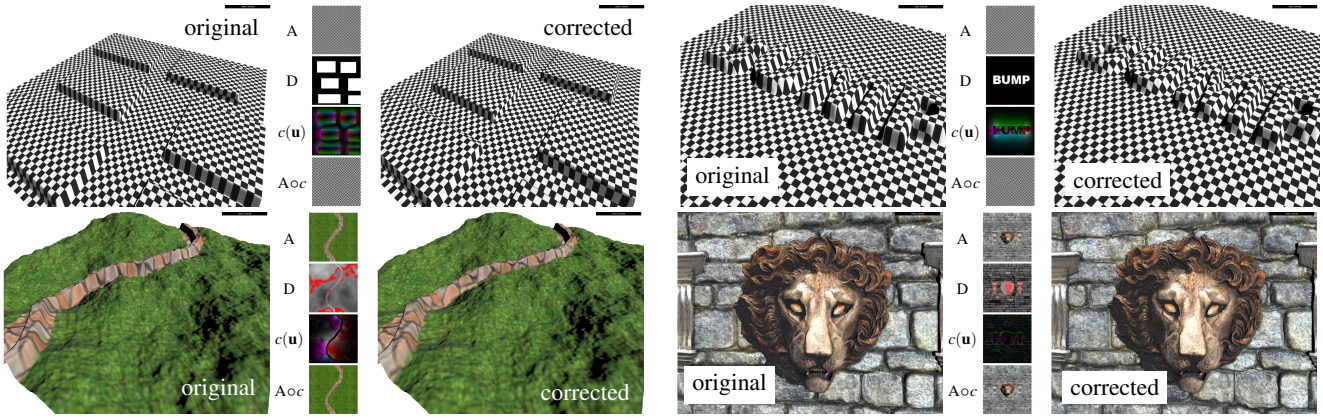

Figure 5: *Our method applied to displacement maps* D *and color maps* A *constructed after or taken from the example cases presented by McGuire et al. [MW08], in order to allow for a more direct comparison of our results with their figure in their paper. Areas that are fixed by correlation constraints are overlain in red on top of the displacement map* D.

4. Results

Results are reported for an Nvidia Geforce GTX 560. Images are shaded with screen-space ambient occlusion and image-based lighting, not included in timings. We also provide a supplemental WebGL demo with full source code. We perform displacement mapping using tessellation shaders [Tat07]. Note that other displacement mapping techniques (such as parallax occlusion mapping [PHL91]) that require to read the displacement many times do not imply that our correction is performed more than once.

Overall we found the run-time performance difference too small to be measured reliably. The offline optimization process takes between one and a few *seconds* on the GPU, at a resolution of 256×256 and 300 iterations, including real-time preview of all preliminary results, see also the supplemental videos. In contrast to previous CPU-based approaches, which take up to several *minutes* [MW08], our method therefore allows for easy integration into interactive content creation tools with potential for real-time user intervention, e. g., for constrained undistortions.

Fig. 6 and Fig. 7 show results of our approach in different scenes. The first column shows common displacement mapping. The second column shows our correction enabled with resampling. The third column show our correction without resampling. The final column visualizes (from top to bottom) the height map used to create **f**, the correction field **c** (where red means horizontal, green indicates vertical correction) and the initial energy **e** (where white indicates a high energy). The supplemental video shows this figure animated, demonstrating convergence of the solver.

Figure 5 provides results for our method in scenes constructed to match the overview figure of McGuire et al. [MW08], in order to allow for a better comparison with their work. Figure 4 shows how our method interacts with minification filtering. When mip maps are used for both the color and the correction offset texture, the result is still well-behaved (also see our WebGL demo showing the effect in motion). Finally, Figure 3 illustrates how undesired distortion caused by our displacement unwarping can be controlled with our optional correlation constraints.

5. Conclusion

In this paper we addressed a common, but besides academic previous work [MW08] to our knowledge not yet often addressed artifact in displacement mapping. We proposed an efficient GPU solution to compute a correction that can be used without knowing the shader to correct and without additional computation.

Our approach allows for fast displacement map correction updates that can provide fast feedback for content creation. Also our correction is not required for all texture maps: Texture maps that were created from a carefully designed automatic process, such as simplification [CMRS98] are not affected much by the distortion described. Correction is most important for stochastic textures showing meso- and micro-structure (stone, bark, sand) acquired form a flat surface.

Since our technique applies to the texture globally, it may move features meant to align with geometric features of the displaced surface (note that this is also true for [MW08]). Further exploration of constraints that preserve correlations between the base geometry and the texture is an interesting avenue for future work.

We proposed two variants that both have benefits and drawbacks. In future work, we would like to devise an on-the-fly correction, a correction with less memory overhead or a mixture between resampling and no-resampling.

References

[BL76] BAJCSY R., LIEBERMAN L.: Texture gradient as a depth cue. *Computer Graphics and Image Processing 5*, 1 (1976), 52–67. 2

[CCC87] COOK R. L., CARPENTER L., CATMULL E.: The Reyes image rendering architecture. *ACM SIGGRAPH Computer Graphics 21*, 4 (1987), 95–102. 1

[CMRS98] CIGNONI P., MONTANI C., ROCCHINI C., SCOPIGNO R.: A general method for preserving attribute values on simplified meshes. In *Proc. IEEE Visualization* (1998), pp. 59–66. 6

[Coo84] COOK R. L.: Shade trees. *ACM SIGGRAPH Computer Graphics 18*, 3 (1984), 223–231. 2

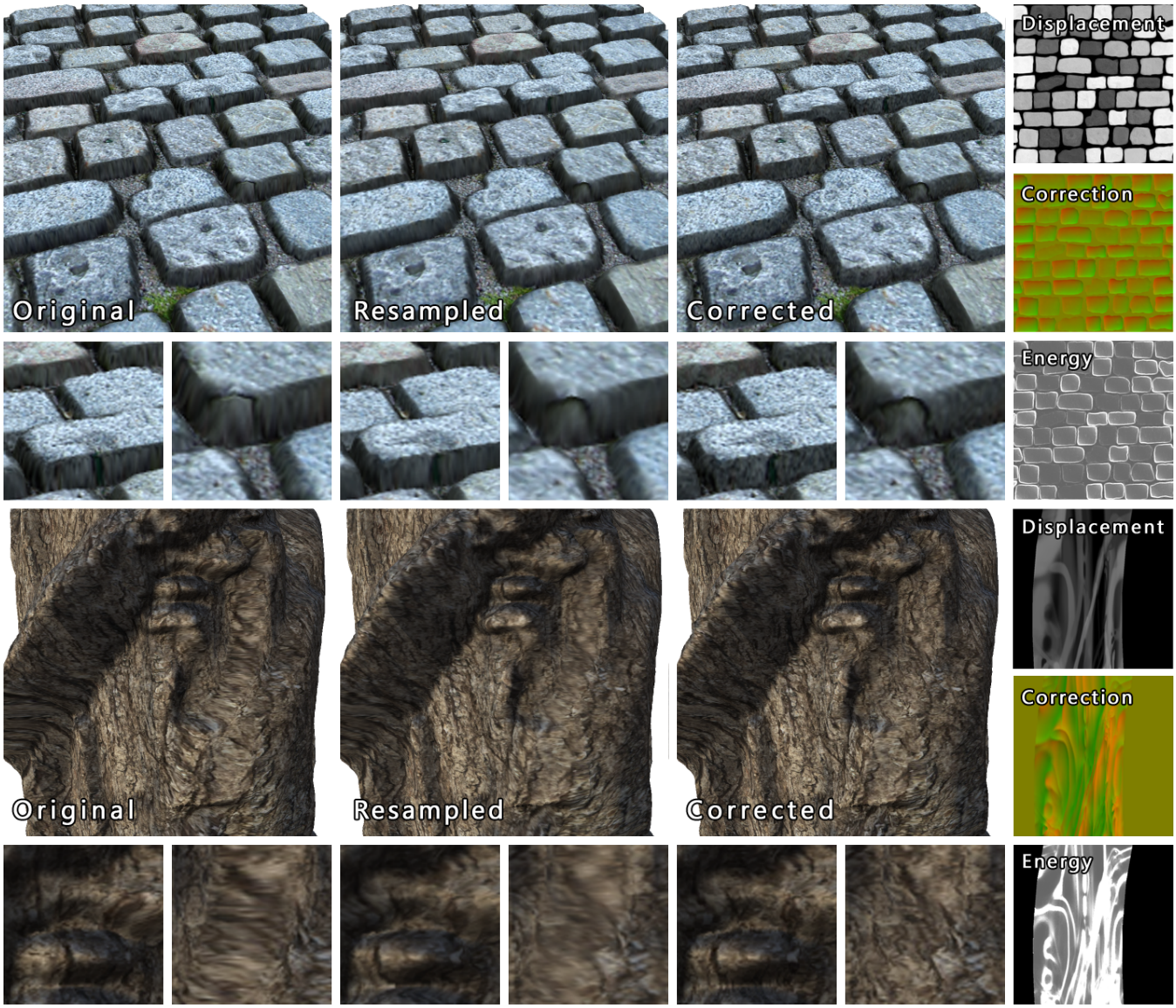

Figure 6: *Results of our approach for two scenes.* Top: *A cobblestone-covered pavement. The displacement map has a resolution of* 512×512 *texels. Solving took* 1 *second with 250 iterations. Common displacement mapping has difficulties at the vertical structures on the stones, where rubber band-like, vertical stripes appear that are not plausible for stone. Both variants of our techniques remove this artifact. The resampling technique performs worse, as there is not enough resolution to resolve the quickly changing vertical structure. In the correction map, it can be seen that correction "pushes" coordinates from constant areas into areas with drastic changes. Not surprising, the energy is highest in areas where the displacement map is discontinuous and proportional to the size of the change.* Bottom: *Bark on a tree trunk (*1024×768 *displacement map, 400 iterations, 2 seconds solving). Common displacement mapping stretches the bark pattern on the surface bumps which is fixed by our approach. The resampling technique performs worse, as there is not enough resolution to resolve the quickly changing vertical structure. The resampling results in quality roughly in-between the common method and our method without resampling.*

[DMK03] DEGENER P., MESETH J., KLEIN R.: An adaptable surface parameterization method. In *Proc. Int. Meshing Roundtable* (2003), pp. 201–213. 2, 3

[FH05] FLOATER M. S., HORMANN K.: Surface parameterization: a tutorial and survey. In *Advances in multiresolution for geometric modelling*. Springer, 2005, pp. 157–186. 2

[GGH02] GU X., GORTLER S. J., HOPPE H.: Geometry images. *ACM Trans. Graph. 21*, 3 (2002), 355–361. 2

[GP09] GONZÁLEZ F., PATOW G.: Continuity mapping for multi-chart textures. *ACM Trans. Graph. 28*, 5 (2009), 109. 4

[Hor01] HORMANN K.: *Theory and Applications of Parameterizing Triangulations*. PhD thesis, Department of Computer Science, University of Erlangen, 2001. 4

[HPS08] HORMANN K., POLTHIER K., SHEFFER A.: Mesh parameterization: Theory and practice. In *ACM SIGGRAPH Asia Course Notes* (2008). 2, 3

[HW62] HUBEL D. H., WIESEL T. N.: Receptive fields, binocular interaction and functional architecture in the cat's visual cortex. *J Physiology*

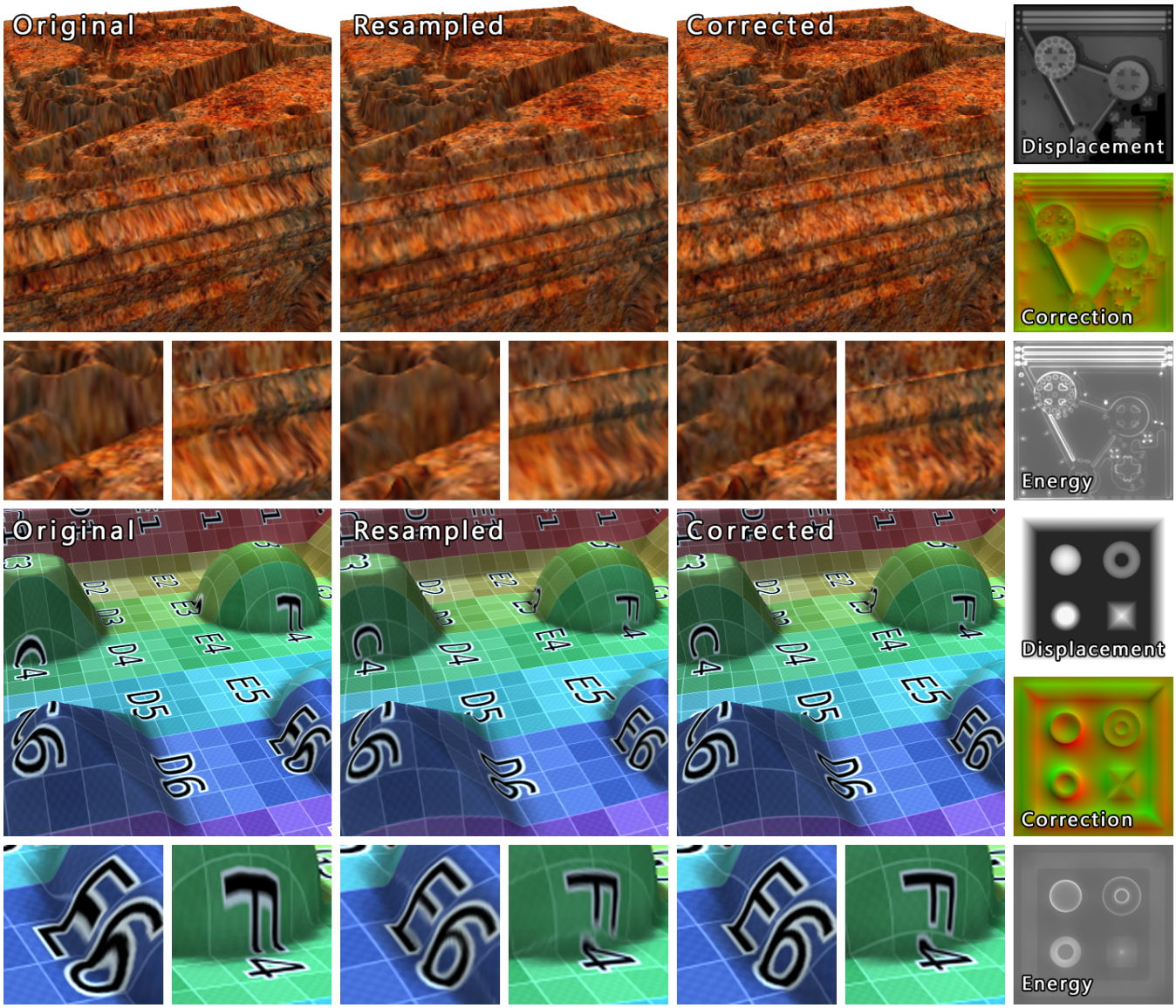

Figure 7: *Results of our approach for two scenes.* Top: *A rusted machine part with geometry details entirely modeled using displacement mapping (*512×512 *displacement map, 400 iterations, 1.5 seconds solving). Differences are best visible in the difficult horizontal structures on the cube corner which incur a substantial distortion. Our approach evenly distributed the rust pattern.* Bottom: *The teaser scene (Fig. 1) from a different view (*256×256 *displacement map, 300 iterations, 0.5 seconds solving). Note, how the patterns used are very common for texture and displacement mapping, and still the artifact is present.*

160, 1 (1962), 106. 2

[MJH*17] MORITZ J., JAMES S., HAINES T. S. F., RITSCHEL T., WEYRICH T.: Texture stationarization: Turning photos into tileable textures. *Comp. Graph. Forum (Proc. Eurographics) 36*, 2 (2017). 2

[MW08] MCGUIRE M., WHITSON K.: Indirection mapping for quasi-conformal relief texturing. In *Proc. i3D* (2008), pp. 191–198. 1, 2, 5, 6

[PHL91] PATTERSON J. W., HOGGAR S. G., LOGIE J.: Inverse displacement mapping. *Comp. Graph. Forum 10*, 2 (1991), 129–39. 2, 6

[RBM06] RITSCHEL T., BOTSCH M., MÜLLER S.: Multiresolution GPU mesh painting. In *EG Short Papers* (2006). 4

[SGSH02] SANDER P. V., GORTLER S. J., SNYDER J., HOPPE H.: Signal-specialized parametrization. In *Proc. EGWR* (2002), pp. 87–98. 2, 5

[SKU08] SZIRMAY-KALOS L., UMENHOFFER T.: Displacement mapping on the GPU – state of the art. *Comp. Graph. Forum 27*, 6 (2008), 1567–92. 2, 3

[SLMB05] SHEFFER A., LÉVY B., MOGILNITSKY M., BOGOMYAKOV A.: ABF++: fast and robust angle based flattening. *ACM Trans. Graph. (TOG) 24*, 2 (2005), 311–30. 2

[SWG*03] SANDER P. V., WOOD Z. J., GORTLER S. J., SNYDER J., HOPPE H.: Multi-chart geometry images. In *Proc. SGP* (2003), pp. 146–155. 2

[Tat07] TATARCHUK N.: Advanced real-time rendering in 3d graphics and games. In *ACM SIGGRAPH Course Notes* (2007). 2, 6

[WTL*04] WANG X., TONG X., LIN S., HU S., GUO B., SHUM H.-Y.: Generalized displacement maps. In *Proc. Rendering Techniques* (2004), pp. 227–33. 2

Appendix A: Undistortion Optimization Code

In this appendix, we provide GLSL fragment shader code that allows for an incremental computation of the relative correction offsets that can then be stored in a texture to allow for lookup of the undistorted texture coordinates. Adding the stored correction offsets to the texture coordinates of the lookup yields the final undistorted texture coordinate.

The GLSL fragment shader code should be run at displacement map resolution, issuing one draw call for each optimization iteration. We use a ping pong buffer comprised of two textures that are alternately set as render target and input texture, in order to compute improved correction offsets from the previous offsets, respectively. Initially, the input correction offset texture is cleared to 0.

```
uniform ivec2 resolution; uniform vec2 pixelWidth;
uniform float areaPreservation;

float areaPreservePow(float x) { return pow(x, areaPreservation); }
float areaPreservePowDeriv(float x) {
  return areaPreservation * pow(x, areaPreservation - 1.0);
}

vec2 computeGradient(vec3 dnpos1, vec3 dnpos2,
  vec2 dnuv1, vec2 dnuv2, out float energy) {
  float adpos = length(cross(dnpos1, dnpos2));
  float aduv = dnuv1.x * dnuv2.y - dnuv2.x * dnuv1.y;

  float energyPart1 = 1.0 / (adpos * aduv);
  float energyPart2 =
      lengthSqared(dnuv1 - dnuv2) * dot(dnpos1, dnpos2)
    + lengthSqared(dnuv1) * dot(dnpos2 - dnpos1, dnpos2)
    + lengthSqared(dnuv2) * dot(dnpos1, dnpos1 - dnpos2);
  float energyPart3Inner = adpos / aduv + aduv / adpos;
  float energyPart3 = areaPreservePow(energyPart3Inner);

  energy = energyPart1 * energyPart2 * energyPart3;
  return
    energyPart3 * (
      (energyPart2 * energyPart1 / aduv)
        * -vec2(dnuv2.y - dnuv1.y, dnuv1.x - dnuv2.x)
      + energyPart1 * 2.0 * (
        (dnuv1) * dot(dnpos2 - dnpos1, dnpos2)
      + (dnuv2) * dot(dnpos1, dnpos1 - dnpos2)
      )
    )
    + energyPart1 * energyPart2 * (
        areaPreservePowDeriv(energyPart3Inner)
      * vec2(dnuv2.y - dnuv1.y, dnuv1.x - dnuv2.x)
      * (1.0 / adpos - adpos / (aduv * aduv))
    );
}

float computeEnergy(vec3 dnpos1, vec3 dnpos2
  , vec2 dnuv1, vec2 dnuv2) {
  float energy;
  computeGradient(dnpos1, dnpos2, dnuv1, dnuv2, energy);
  return energy;
}

vec2 computeConstraintGradient(vec2 duv, float pinned, out float energy) {
  vec2 mc = pinned * vec2(resolution);
  float ec = 1.0 / max(1.0 - lengthSqared(duv * mc), 0.5e-32);
  energy = 0.5 * ec;
  return ec * ec * (duv * mc * mc);
}

bool onGlobalBorder(ivec2 c) {
  return c.x==0||c.y==0||c.x==resolution.x-1||c.y==resolution.y-1;
}
```

Listing 1: *GLSL code computing E_i and $\nabla \mathbf{e}_i$.*

```
uniform sampler2D displaceTex;
uniform sampler2D correctionOffsetTex;
uniform float texelOffset;

uniform float displacementScale;
uniform float fixHeight, fixInterval; uniform bool fixBoundary;

struct Node {
  vec3 pos;
  vec2 uv;
  vec2 uvo;
  float pinned;
};
Node fetchNode(vec2 coord) {
  Node n;
  coord *= pixelWidth;
  vec2 wrapCoord = fract(coord);
  // apply half texel offset to match geometric displacement
  float height = textureLod(displaceTex
    , wrapCoord - texelOffset * pixelWidth, 0.0).x;
  n.pos = vec3(coord, displacementScale * height);
  n.uvo = textureLod(correctionOffsetTex, wrapCoord, 0.0).xy;
  // compute constraints from given height interval
  n.pinned = (fixInterval > 0.0)
    ? max(1.0 - abs(height - fixHeight) / fixInterval, 0.0) : 0.0;
  n.uv = n.uvo + coord;
  return n;
}

uniform int iterationIdx;
out vec4 newCorrectionOffset;
```

Listing 2: *GLSL code for input and output data.*

```
ivec2 currentCoord = ivec2(gl_FragCoord.xy);
Node currentNode = fetchNode(gl_FragCoord.xy);

#define NUM_NEIGHBORS 6
vec3 dnpos[NUM_NEIGHBORS];
vec2 dnuv[NUM_NEIGHBORS];
{ int i = 0;
  for (ivec2 co = ivec2(-1); co.y <= 1; ++co.y) {
    for (co.x = -1; co.x <= 1; ++co.x) {
      if (co.x != co.y) {
        Node nn = fetchNode(gl_FragCoord.xy + vec2(co));
        // compute the neighbor index
        int nbidx = i; // i = 0,1
        if (co.y == 0) nbidx = (co.x == -1) ? 5 : 2;
        else if (i >= 4) nbidx = 8 - i; // i = 4,5 -> nbidx = 4,3
        // compute the neighbor offsets
        dnpos[nbidx] = currentNode.pos - nn.pos;
        dnuv[nbidx] = currentNode.uv - nn.uv;
        ++i;
      }
    }
  }
}

float energy = 0.0; // in case you want to see it
vec2 gradient = vec2(0.0);
// compute energy gradient for each neighbor triangle
for (int i = 0, j = NUM_NEIGHBORS - 1; i < NUM_NEIGHBORS; j = i++) {
  float neighborEnergy;
  gradient += computeGradient(
    dnpos[j], dnpos[i], dnuv[j], dnuv[i], neighborEnergy);
  energy += neighborEnergy;
}
energy *= 0.5 / float(NUM_NEIGHBORS);

// handle constrained nodes
{
  float cEnergy;
  gradient += computeConstraintGradient(
    currentNode.uv - currentNode.pos.xy, currentNode.pinned, cEnergy);
  energy += cEnergy;
}
```

Listing 3: *GLSL code computing the optimization gradients.*

```
float minStepSize = 0.0;
float maxStepSize = 2.0e32;
// compute maximum step size (before triangle flips)
for (int i = 0, j = NUM_NEIGHBORS - 1; i < NUM_NEIGHBORS; j = i++) {
  float aduv = dnuv[j].x  * dnuv[i].y  - dnuv[i].x  * dnuv[j].y;
  float djXg = dnuv[j].x  * gradient.y - gradient.x * dnuv[j].y;
  float dgXi = gradient.x * dnuv[i].y  - dnuv[i].x  * gradient.y;
  float den = djXg + dgXi;
  float aduv0offset = aduv / den;
  if (aduv0offset > 0.0) {
    if (aduv > 0.0)
      maxStepSize = min(maxStepSize, aduv0offset);
    else
      minStepSize = max(minStepSize, aduv0offset);
  }
}
// assert: minStepSize <= maxStepSize
if (!(minStepSize <= maxStepSize))
  minStepSize = maxStepSize = 0.0;

// Perform ternary search on gradient line to find optimum
while (maxStepSize - minStepSize > 1.0e-6 * maxStepSize) {
  float third1 = mix(minStepSize, maxStepSize, 1.0 / 3.0);
  float third2 = mix(minStepSize, maxStepSize, 2.0 / 3.0);
  vec2 duv1 = third1 * gradient, duv2 = third2 * gradient;

  float e1 = 0.0, e2 = 0.0;
  for (int j = NUM_NEIGHBORS - 1, i = 0; i < NUM_NEIGHBORS; j = i++) {
    e1 += computeEnergy(dnpos[j],dnpos[i],dnuv[j]-duv1,dnuv[i]-duv1);
    e2 += computeEnergy(dnpos[j],dnpos[i],dnuv[j]-duv2,dnuv[i]-duv2);
  }
  e1 *= 0.5 / float(NUM_NEIGHBORS);
  e2 *= 0.5 / float(NUM_NEIGHBORS);
  float pe1, pe2;
  computeConstraintGradient(
    currentNode.uv - duv1 - currentNode.pos.xy, currentNode.pinned, pe1);
  computeConstraintGradient(
    currentNode.uv - duv2 - currentNode.pos.xy, currentNode.pinned, pe2);
  e1 += pe1; e2 += pe2;
  if (e1 > e2) minStepSize = third1;
  else maxStepSize = third2;
}
float stepSize = mix(minStepSize, maxStepSize, 0.5);

// boundary handling
if (fixBoundary && onGlobalBorder(currentCoord)) stepSize = 0.0;

newCorrectionOffset = currentNode.uvo.xyxy;
// ensure convergence by only ever moving 1 vertex per neighborhood
if (   (currentCoord.x & 1) == (iterationIdx & 1)
  && (currentCoord.y & 1) == ((iterationIdx >> 1) & 1) )
  newCorrectionOffset.xy -= stepSize * gradient;
```

Listing 4: *GLSL code computing the update steps for the iterative optimization of undistortion correction offsets.*