Analytic Displacement Mapping using Hardware Tessellation

Matthias Nießner
University of Erlangen-Nuremberg
and
Charles Loop
Microsoft Research

Displacement mapping is ideal for modern GPUs since it enables high-frequency geometric surface detail on models with low memory I/O. However, problems such as texture seams, normal re-computation, and under-sampling artifacts have limited its adoption. We provide a comprehensive solution to these problems by introducing a smooth analytic displacement function. Coefficients are stored in a GPU-friendly tile based texture format, and a multi-resolution mip hierarchy of this function is formed. We propose a novel level-of-detail scheme by computing per vertex adaptive tessellation factors and select the appropriate pre-filtered mip levels of the displacement function. Our method obviates the need for a pre-computed normal map since normals are directly derived from the displacements. Thus, we are able to perform authoring and rendering simultaneously without typical displacement map extraction from a dense triangle mesh. This not only is more flexible than the traditional combination of discrete displacements and normal maps, but also provides faster runtime due to reduced memory I/O.

Categories and Subject Descriptors: I.3.7 [**Computer Graphics**]: Picture/Image Generation—*Display Algorithms*; I.3.5 [**Computer Graphics**]: Computational Geometry and Object Modeling—*Curve, surface, solid, and object representations*

General Terms: Subdivision Surfaces, GPU

Additional Key Words and Phrases: Catmull-Clark subdivision surfaces, displacement mapping, hardware tessellation

ACM Reference Format:

Pamplona, V. F., Oliveira, M. M., and Baranoski, G. V. G. 2009. Photorealistic models for pupil light reflex and iridal pattern deformation. ACM Trans. Graph. 28, 4, Article 106 (August 2009), 11 pages.
DOI = 10.1145/1559755.1559763
http://doi.acm.org/10.1145/1559755.1559763

Manuel M. Oliveira acknowledges a CNPq-Brazil fellowship (305613/2007-3). Gladimir V. G. Baranoski acknowledges a NSERC-Canada grant (238337). Microsoft Brazil provided additional support.
Authors' addresses: land and/or email addresses.

1. INTRODUCTION

Displacement mapping has been used as a means of efficiently representing and animating 3D objects with high frequency surface detail. Where texture mapping assigns color to surface points at u, v parameter values, displacement mapping assigns vector offsets. The advantages of this approach are two-fold. First, only the vertices of a coarse (low frequency) base mesh need to be updated each frame to animate the model. Second, since the only connectivity data needed is for the coarse base mesh, significantly less space is needed to store the equivalent highly detailed mesh. Further space reductions are realized by storing scalar, rather than vector offsets. The displacement is then achieved by offsetting a base surface point in its normal direction according to the value stored in a scalar displacement map. While not as flexible from a modeling standpoint as vector displacement mapping, scalar displacement mapping significantly reduces data throughput in the graphics pipeline, as well as the overall storage space and transmission requirements for digital models.

Recently introduced hardware tessellation is ideally suited to displacement mapping. Higher order parametric patches provide a base surface that is evaluated on-chip to form a dense triangle mesh and immediately rasterized with low memory I/O. Displacing triangle vertices in their normal direction according to a value stored in texture memory has very little performance impact. However, while conceptually simple and highly efficient, hardware displacement mapping has not been widely adopted in real-time applications due to several subtle artifacts. We address these artifacts in this paper.

1.1 Displacement Mapping Artifacts

Meshes are typically endowed with a parameterization in the form of a 2D texture atlas. Conceptually, a few seams are introduced on edges to *unfold* the surface into the plane, creating a mapping (an atlas) from the plane to the surface. Points on seams map to more than one point in texture space resulting in inconsistent values; bilinear texture filtering exacerbates this problem. For displacement mapping, this can lead to unacceptable cracks in a rendered surface.

The normal of the base surface serves as the direction of displacement. However, this base surface normal is, in general, *not* the normal of the resulting displaced surface; complicating accurate shading. To overcome this problem, *normal mapping* has been used to assign more plausible surface normals over displaced vertices to reduce shading artifacts. To allow the base surface to be deformed, *tangent space* normal mapping is used, where the xyz coordinates of the normal relative to a tangent frame are stored. The computation of tangent frames is costly and technically challenging since these must be globally consistent across mesh edges. While the resulting shading is often plausible, the deformed normal field does not correspond to the displaced surface; hence it is not accurate. Furthermore, re-computation of normal maps on-the-fly

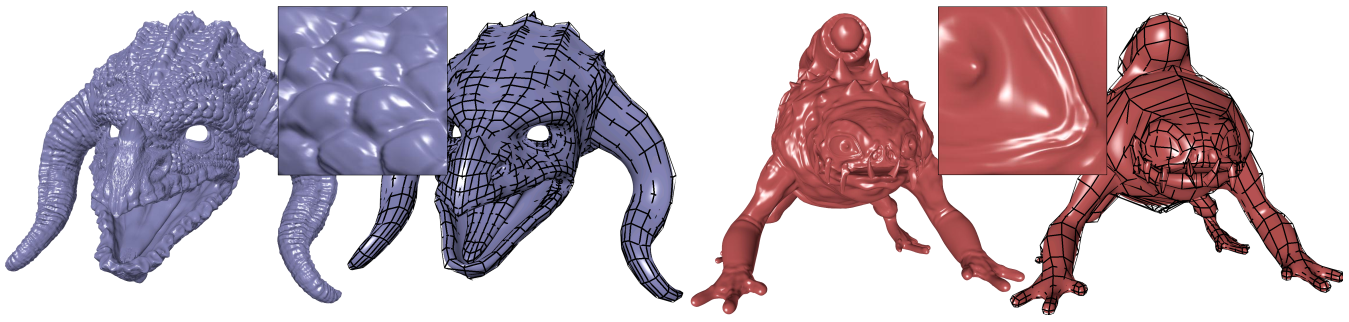


Fig. 1. The Dragon Head (2706 patches; 1.7 ms and 1.2 ms) and Monster Frog (1292 patches; 1.3 ms and 0.85 ms) model rendered w/ and w/o our displacement method. We displace vertices according to an analytic displacement function on top of a Catmull-Clark base surface using hardware tessellation with dynamically computed tess factors. Since normals are computed based solely on the displacement function, no normal map is required. This allows modifying displacements at runtime and increases performance due to a reduced memory I/O. The models are courtesy of the Blender Foundation and Bay Raitt, respectively.

is necessary at displacement authoring time to give instant feedback. Finally, normal map textures add significantly to the storage and data throughput costs of models.

Hardware tessellation is based on the idea of dynamic re-tessellation of patches. That is, the underlying sampling pattern of patch vertices should be updated every frame to keep the resulting triangle sizes just right; not too small or rasterization becomes inefficient, and not too big so that faceting and interpolation artifacts are kept to a minimum. However, changing this sampling pattern creates *swimming* artifacts in the displaced surface; the surface appears to fluctuate wildly as the sampling pattern changes. This artifact is caused by under-sampling the displacement map.

1.2 Solutions and Contributions

We propose solutions, in the context of displaced Catmull-Clark subdivision surfaces, to all of the artifacts just mentioned. Following Lee et al. [2000], we write the displaced surface as

$$f(u,v) = s(u,v) + N_s(u,v)D(u,v), \qquad (1)$$

where $s(u,v)$ is a base Catmull-Clark limit surface defined by a coarse base mesh, $N_s(u,v)$ is its corresponding normal field, and $D(u,v)$ is a scalar valued displacement function. We chose Catmull-Clark since it is an industry standard, but our ideas could be extended to Loop subdivision as well; the important property we leverage is that the base surface is everywhere C^2, except at a limited number of *extraordinary vertices* where it is still C^1. Requiring the base surface $s(u,v)$ to be C^2 ensures that its normal field $N_s(u,v)$ will be C^1. Furthermore, by constructing the displacement function $D(u,v)$ to be C^1 with vanishing first derivatives at extraordinary vertices, we can guarantee that the displaced surface $f(u,v)$ will be C^1 smooth everywhere.

Our displacement function $D(u,v)$ is a scalar valued biquadratic B-spline surface with a Doo-Sabin subdivision surface structure. Therefore, the discrete values found in our displacement maps are the coefficients of this surface. Our motivation in making this choice was to minimize the cost of (per-pixel) evaluation while also providing a C^1 smooth displacement function; biquadratic splines are optimal for this.

In order to deal with the problem of texture seam misalignment, we devise a tile based texture format, similar to Ptex [Burley and Lacewell 2008], that corresponds to the quad faces of the base mesh (possibly after one level of local subdivision). Unlike Ptex, our format is specifically designed for the GPU, and we eliminate the neighbor face pointers that hampers its data parallel implementation. We also deal gracefully with non-uniform tile sizes so that surface detail is appropriately distributed over a surface. This includes down-sampled displacements (i.e., mip levels) while providing matching displacements at tile boundaries.

Since the position and derivatives of the displaced surface $f(u,v)$ can be evaluated analytically, no normal maps are required. Furthermore, we demonstrate an evaluation procedure where the low frequency base surface $s(u,v)$ is evaluated at triangle vertices in the domain shader, and the derivatives of the high frequency displacement function $D(u,v)$ are evaluated in the pixel shader; resulting in highly accurate surface shading, even during animation. Our scheme also supports *dynamic* displacement mapping; where the displacement function can change at runtime.

Finally, we provide a novel and efficient level-of-detail scheme based on a multi-resolution analysis of the displacement function $D(u,v)$. This includes computing the tessellation density on-the-fly and selecting the appropriate mip level of the displacements. This allows us to avoid the under-sampling problems that cause the swimming artifacts typically encountered when varying hardware tessellation factors. Furthermore, we are able to eliminate popping artifacts by employing fractional tessellation factors and filtering between corresponding mip levels.

2. PREVIOUS WORK

Texturing: Parameterizing polygon meshes is a well-known issue in computer graphics. Texture atlases are widely used; however, providing consistent values across chart boundaries is challenging ([Sander et al. 2003], [González and Patow 2009]). The resulting minor color errors are often tolerable in the context of texture mapping; the resulting cracks in the context of displacement mapping are not. Purnomo et al. [2004] address these color errors by finding quadrilateral regions that are aligned in texture space. They compute boundary overlap to obtain seamless texturing, and they propose several strategies to access texture entries. Our approach is similar, but an important difference is where a power-of-two size constraint for tile edges is enforced. Ours is on the interior of tiles (excluding overlap); theirs is on the entire tile (including overlap). While their design choice enables perfect packing (i.e., no wasted space), ours allows for ideal mip pyramids; at a cost of some unused texture space.

Ray et al. [2010] introduce *Invisible Seams*, a method for providing consistent texture accesses and mip mapping using a traditional texture atlas approach. Therefore, they use a set of contraints that are applied after texture atlas updates in order to keep texture data consistent. In contrast to their approach we avoid the use of a global parameterization (u, v atlas) and do not require explicit texture coordinates. Instead, we propose a tile based approach that uses (virtual) implicit texture coordinates that align with the parameterization that comes from the underlying Catmull-Clark base surface. This alignment is essential for our analytic displacement function since it allows us to derive surface normals of the displaced geometry; this would not be feasible with a texture atlas based technique.

Burley and Lacewell [2008] developed Ptex, a tile based texturing format, for off-line rendering, and use adjacency pointers to access neighboring tiles. These pointers must be dereferenced when filtering on tile boundaries. We achieve the same result by storing overlap texels along tile boundaries that correspond to the respective texels of neighboring tiles. While Ptex does not pre-filter tile data, we form full mip pyramids over tiles to accelerate level-of-detail management and avoid under-sampling. Mesh colors [Yuksel et al. 2010] is another per-face texturing method. Instead of overlap or adjacency pointers, Mesh colors stores data indices in texture maps; consistency is achieved by index sharing. However, this scheme requires an extra level of indirection that reduces performance, particularly on the GPU.

Displacement Mapping: Blinn [1978] proposed perturbing surface normals using a wrinkle function. While this mimics the shading of a high resolution surface, the geometry itself remains unchanged. This lead Cook [1984] to develop displacement mapping in order to give objects more realistic silhouettes. The use of scalar displacements in the context of multi-resolution modeling has been proposed by Guskov et al. [2000]. Lee et al. [2000] use a similar idea, but they apply displacements on top of a Loop [1987] subdivision surface. Additionally, they obtain the displacement function and its derivatives via costly iterative subdivision; our approach involves direct evaluation. Mapping discrete displacement values on Catmull-Clark subdivision surfaces was proposed by Bunnell [2005]. We also use Catmull-Clark [1978] as a base surface, but apply a displacement function using biquadratic B-splines with a Doo-Sabin [1978] subdivision structure. An overview of traditional displacement mapping approaches on the GPU is given by Szirmay-Kalos and Umenhoffer [2008]. Implementing displacement mapping in the context of hardware tessellation is shown by Tatarchuck et al. [2010]. Schäfer et al. [2012] also use the tessellator to apply displacements. They assign vertex attributes (such as displacement values) in the domain shader using a shared index format similar to Mesh colors.

Subdivision Surfaces on the GPU: Rendering subdivision surfaces on the GPU has been done iteratively, by repeated mesh refinement to temporary buffers, and then drawing the resulting triangle mesh [Bunnell 2005],[Shiue et al. 2005],[Patney et al. 2009]. While this could be combined with displacement mapping, the significant I/O of streaming data to and from the GPU memory limits performance. On modern GPUs hardware tessellation enables evaluating and rendering the surface on-chip without these costly memory transfers. Hardware tessellation requires direct surface evaluation rather than the iterative application of subdivision rules. This can be realized using approximate Catmull-Clark subdivision methods (cf. [Myles et al. 2008], [Loop et al. 2009]). These G^1 approximations are not adequate for our purposes since the resulting displaced surface will not be analytically smooth. Exact evaluation using Stam's [1998] direct approach is possible, but slow (due to significant code branching, eigenbasis coefficient lookup, and floating point computation). Nießner et al. [2012] perform adaptive subdivision around extraordinary vertices using GPGPU compute kernels and process the resulting bicubic patches with the hardware tessellator. This method is also exact and relatively fast. We use their approach to evaluate our Catmull-Clark base surface.

3. ALGORITHM OVERVIEW

Our base surface $s(u, v)$ is the limit surface of Catmull-Clark subdivision defined by a two-manifold control mesh, possibly with boundaries. While this surface is traditionally defined as the result of the repeated application of a set of subdivision rules, we (following [Halstead et al. 1993], [Stam 1998], and [Nießner et al. 2012]) treat this surface in a parametric form. The topology, geometry, and parameterization of this surface are characterized by its defining control mesh. If the faces of the control mesh are not exclusively quadrilateral, then one refinement step will ensure this. A one-to-one correspondence between these quadrilateral faces and unit square domains is established, giving rise to a global parameterization of the surface (via a face ID; $u, v \in [1, 0] \times [0, 1]$ triple). This is defined by the corresponding subdivision of quadrilateral control mesh faces and unit square domains. This process has well-defined limits and yields a closed form (via eigenbasis functions [Stam 1998], or bicubic subpatches [Nießner et al. 2012]). In the interest of simplicity, we assume consistency of quad face ID and unit square domain ID; we therefore safely exclude this book keeping detail from our notation.

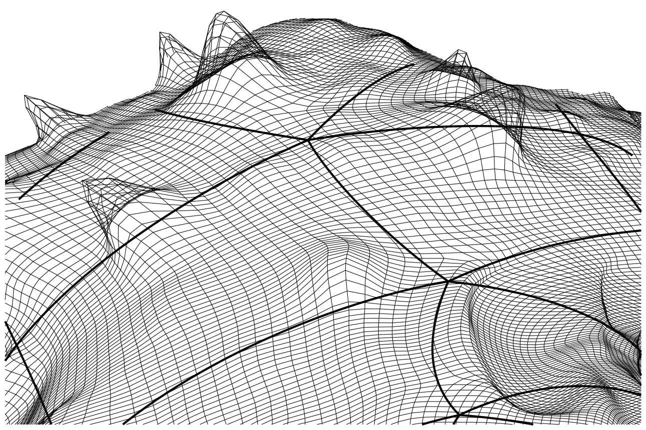


Fig. 2. Base surface: Catmull-Clark limit patches - patch boundaries shown as thick lines. Displacement surface: Biquadratic Doo-Sabin B-splines - scalar coefficients on top of base surface normal field, shown as thin lines.

For our analytic displacement function $D(u, v)$, we use biquadratic B-splines. These patches have an arrangement that is consistent with Doo-Sabin subdivision. This means that the control mesh for our displacement coefficients is *dual*, with refinements, to the control mesh of the base mesh. Note however, that $D(u, v)$ is scalar valued and can be thought of as a *height field*. In other words, both the base surface $s(u, v)$ and the displacement function $D(u, v)$ correspond to the same topological two-manifold; though embedded in $\mathbb{R}^3$ and $\mathbb{R}^1$ respectively. Note again, that choosing biquadratic B-splines closely related to Doo-Sabin subdivision gives us a globally C^1 displacement function that is less costly to compute than higher order alternatives.

Figure 2 shows a detail view of a model with base patch edges (thick curves) and the displacement function coefficients over the base surface (thin grid). As a practical matter, we deal with extraordinary vertices by imposing a constraint that causes first derivatives of the displacement function $D(u, v)$ to vanish at these points. This degeneracy means that $D(u, v)$ is a globally C^1 function that can be evaluated over the entire manifold without special case handling, see Section 4.2 for a detailed explanation.

For each quad face of the base surface control mesh, we define a *texture tile* that contains the coefficients of the displacement function. For non-quad faces, we locally subdivide once to obtain quads. To evaluate the displacement function near tile boundaries, we need coefficient data from adjacent tiles. Trying to explicitly access adjacent tile data on the GPU would degrade performance since only boundary evaluations would require this, causing a branch and breaking data parallelism. Instead, we pad our tiles with a one texel overlap region; this ensures good data parallel performance since boundary evaluations will not be a special case. We devise a straightforward tile based texture format in Section 4 that contains these overlaps, as well as a simple solution to tile access issues caused by extraordinary vertices.

Treating the displaced surface in a smooth analytic form means that it will have a well-defined, smooth normal field; this will eliminate many shading artifacts. Furthermore, the separation of the displaced surface into a low frequency base surface and a high frequency displacement function is ideally suited to a modern graphics pipeline implementation. We evaluated the base surface and its partial derivatives at a relatively low frequency in the domain shader. The derivatives of the displacement function are then evaluated at a higher frequency in the pixel shader. The interpolated low frequency data, combined with the evaluated high frequency data results in a highly accurate normal field, ideal for lighting calculations (see Section 5).

To deal with swimming artifacts caused by displacement undersampling, we form mip pyramids over texture tiles via a multiresolution analysis of the displacement function $D(u, v)$. In order to remove swimming artifacts, we employ a smooth level-of-detail scheme that matches sampling density to the appropriate displacement function mip level (see Section 6).

Figure 3 provides an overview of our rendering algorithm.

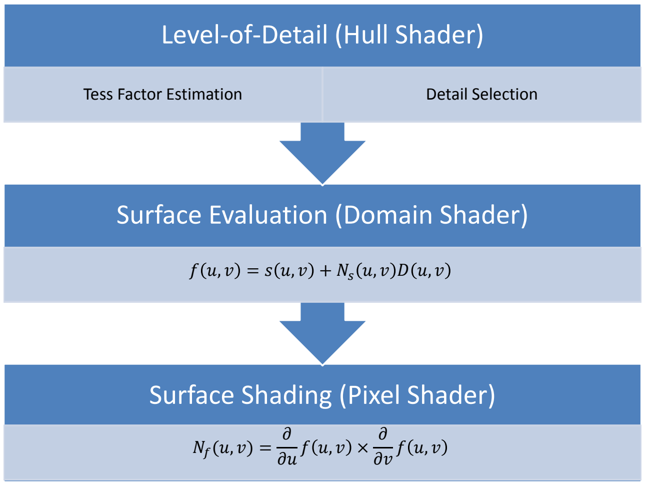

Level-of-Detail (Hull Shader)
Tess Factor Estimation
Detail Selection
Surface Evaluation (Domain Shader)
$f(u,v) = s(u,v) + N_s(u,v)D(u,v)$
Surface Shading (Pixel Shader)
$N_f(u,v) = \frac{\partial}{\partial u} f(u,v) \times \frac{\partial}{\partial v} f(u,v)$

Fig. 3. Algorithm overview.

4. TILE BASED TEXTURE FORMAT

We store our biquadratic displacement function coefficient data in an axis-aligned tile based texture format. This avoids seam misalignment problems that plague classic u, v atlas parameterization texture methods. Our format can be seen as a GPU version of Ptex [Burley and Lacewell 2008], however, we do not rely on adjacent tile pointers since these are impractical on the GPU. Instead, we store a one texel overlap per tile to enable filtering while matching displacements at tile boundaries. Two of these tiles are shown in Figure 4. Overlap computation, particularly at extraordinary vertices, is described in Section 4.2. Each tile corresponds to a quad face of the Catmull-Clark control mesh. We require tile edges to be a power-of-two plus overlap in size; that is for a $\mathit{tileSize} = 2^k$ (for integer $k \geq 1$) , tile edge lengths are of the form $\mathit{tileSize} + 2$. However, adjacent tiles do not need to be the same size. We currently only support square tiles; but rectangular tiles could be supported at a cost of a few additional storage bits per tile.

4.1 Displacement Data Generation

The base surface and displacement function needed by our algorithm could be generated by a conversion process from a scanned dense triangle mesh, or directly authored using a sculpting tool. Our work is agnostic to this choice, but we discuss the tradeoffs here for the sake of completeness.

Lee et al. [2000] assume that a high resolution triangle mesh is given, and then simplified to obtain a coarse (Loop subdivision) base mesh. The displacement data are then found by extraction using ray casting. Rays are fired for each tile entry from the base mesh in the normal direction and intersected with the (high-resolution) source mesh. Unfortunately, extraction using ray casting has problems. In particular, rays can miss the source surface, and typically requires manual adjustments. These geometry processing issues remain as open research problems that we do not address in this paper. However, assuming *clean* displaced sample data at all tile locations, we are able to convert surfaces with traditional displacements into our tile based format. Therefore, we resample the displacement map and solve for the biquadratic B-spline coefficients to interpolate the displacement data. We have performed this conversion process to generate the displacement data for the sample models Dragon Head and Monster Frog (see Figure 1).

An alternative approach is to integrate scalar displacement modeling into the authoring tool. This implies that sculpting data are restricted to translations along normals of the base surface. Artists can then directly create the displaced surface exactly as it will appear in the final application. The tile based data format we provide in this paper allows a user to directly *paint* on the surface and modify displacements in place. While we apply edits on the CPU, we only update modified tiles in GPU memory in order to keep CPU-to-GPU memory transfer small. Modeling is analogous to multi-resolution editing as used by typical sculpting tools such as MudBox or ZBrush.

We do not claim that the work-flow shown in our authoring tool is necessarily original (details of how professional authoring tools work are proprietary). We do claim however, that by using the techniques described in this paper, the delay and limitations imposed by the intermediate step of dense triangle mesh creation can be avoided. Far less GPU memory is needed, and the model is readily animated without any internal conversions. Furthermore, the analytic nature of our method provides for correct shading at all scales.

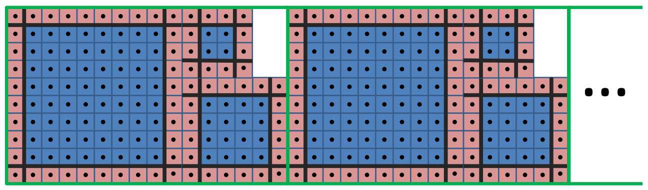


Fig. 4. Snippet of our texture format used for displacement values (8×8 per tile; blue) showing two tiles (bordered green) including overlap (red) and mip levels.

4.2 Overlap at Extraordinary Vertices

Near extraordinary vertices, where more (or less) than four tiles meet, we need a way to efficiently evaluate our displacement function. Our scalar displacement spline has a Doo-Sabin subdivision structure; however, a direct evaluation approach based on eigenbasis functions does not exist since the subdivision matrix for this case is defective [Stam 1998].

Instead, we impose a constraint that will allow us to evaluate the displacement function $D(u, v)$ as a standard biquadratic B-spline over its entire domain. The idea is to set all tile corners corresponding to the same extraordinary vertex to the same value. We find this value by averaging. The result is that $\frac{\partial}{\partial u} D = \frac{\partial}{\partial v} D = 0$ at these tile corners. While this limitation is unfortunate from a modeling perspective, it is beneficial from a rendering perspective. Evaluation of the displacement function $D(u, v)$ is fast and consistent since extraordinary vertices do not require branching to specialize code. Furthermore, we can guarantee that our displacement spline $D(u, v)$, will be C^1 across tile boundaries, for proof see [Reif 1997]. This means that extraordinary vertices will not cause any shading discontinuities.

4.3 Mip Levels and Global Texture Design

The swimming artifacts associated with the dynamic tessellation patterns generated by the hardware tessellation unit are undersampling artifacts. That is, the underlying displaced surface is a high frequency signal that is sampled below its Nyquist rate by the tessellator. This is a classic problem in other context within computer graphics and signal processing that can effectively be resolved using mip mapping [Williams 1983]. Therefore, we precompute a full mip map pyramid for each tile. To generate these mip levels, we tried both Haar wavelets and a wavelet based on quadratic B-splines, so-called *B-wavelets*. Haar wavelets correspond to classic 4-way averaging to down-sample mips levels. The quadratic B-wavelets we used are based on the work of Bertram et al. [2004] and involve a kernel with larger support.

Once all mip levels have been generated, we pack all tiles including its mip levels in a single texture. In order to leverage cache coherence, mip levels of individual tiles are stored next to each other (see Figure 4). While this leaves some unused space, our experiments show that it provides superior performance. In the end we require $(1.5 \cdot \mathit{tileSize} + 4) \cdot (\mathit{tileSize} + 2)$ texture entries for a single tile including overlap and mip levels. Additionally, for each quad face we must store its tile size and an offset to the tile location within the global texture in a separate buffer. Tile data is then indexed by the face ID. Note that local mip level offsets (within a tile) are computed at runtime and do not require additional storage.

4.4 Non-uniform Tile Sizes

We support distinct tile sizes in order to allow localized detail within a mesh. As a result there may be adjacent tiles with distinct resolutions. To avoid cracks, we must ensure the consistency of data accessed along boundaries between mixed resolution tiles. To this end, we require that coarser mip levels of a higher resolution tile correspond to its lower resolution neighbor. This is achieved by computing tile overlap for each mip level separately at matching resolutions. Since not all tiles have the same number of mip levels (i.e., they have different resolutions), there are boundaries (at particular mip levels) where overlap computation cannot be performed.

Tile resolution is characterized by k, where $\mathit{tileSize} = 2^k$ (tile edge length, not including overlap). For each base mesh control vertex, we determine the incident tile with the highest resolution k and use this number k as a base value for that vertex. We then find the differences between the base value of a vertex, and each incident tile's highest resolution (one of these is guaranteed to be zero); we store these differences for each of the four tile corners. These values are packed into a single 32 bit integer stored per tile (see Section 5.4).

At runtime we bilinearly interpolate these difference values for a given u, v parameter value. The tile's own resolution minus this interpolated value tells us the maximum (possibly fractional) mip level that can be accessed for that parameter value. Along edges between mixed resolution tiles, we will always obtain a consistent maximum mip level, and hence consistent data accesses. In Section 6 we discuss a vertex based level-of-detail scheme and how mip levels are selected at runtime.

5. SURFACE RENDERING

Hardware tessellation generates triangle meshes on-the-fly by sampling a user defined evaluation procedure for a parametric surface patch. This paradigm is not compatible with the traditional recursive refinement construction of subdivision surfaces. Several G^1 patch based approximate Catmull-Clark schemes have appeared in recent years (cf. [Myles et al. 2008], [Loop et al. 2009]) to overcome this difficulty. However, these are not adequate for our purposes, since we require a C^2 base surface (in order to guarantee that the final displaced surface is C^1). The direct evaluation procedure from Stam [1998] could be used to evaluate the base surface. However, our experiments on the GPU indicate that *feature adaptive subdivision* described by Nießner et al. [2012] performs significantly better (see Section 5.4).

5.1 Surface Evaluation

For given u, v coordinates and face ID, we evaluate the displaced surface

$$f(u, v) = s(u, v) + N_s(u, v)D(u, v),$$

corresponding to a texture tile, by evaluating the base patch $s(u, v)$, its normal $N_s(u, v)$, and the corresponding displacement function $D(u, v)$. The scalar displacement function is evaluated by selecting the 3×3 array of coefficients for the biquadratic subpatch of $D(u, v)$, corresponding to the u, v value within its tile domain. We transform the patch parameters u, v into the subpatch domain $(\hat{u}, \hat{v})$ using the linear transformation T:

$$\hat{u} = T(u) = u - \lfloor u \rfloor + \tfrac{1}{2}, \text{ and } \hat{v} = T(v) = v - \lfloor v \rfloor + \tfrac{1}{2}.$$

We then evaluate the scalar displacement function

$$D(u,v) = \sum_{i=0}^{2}\sum_{j=0}^{2} B_i^2(T(u))B_j^2(T(v))d_{i,j},$$

where $d_{i,j}$ are the selected displacement coefficients, and $B_i^2(u)$ are the quadratic B-spline basis functions.

The base surface normal $N_s(u,v)$ is obtained from the partial derivatives of $s(u,v)$:

$$N_s(u,v) = \frac{\frac{\partial}{\partial u}s(u,v) \times \frac{\partial}{\partial v}s(u,v)}{\left\|\frac{\partial}{\partial u}s(u,v) \times \frac{\partial}{\partial v}s(u,v)\right\|_2}.$$

In order to obtain the normal of the displaced surface of $f(u,v)$, we compute its partial derivatives:

$$\frac{\partial}{\partial u}f(u,v) = \\ \frac{\partial}{\partial u}s(u,v) + \frac{\partial}{\partial u}N_s(u,v)D(u,v) + N_s(u,v)\frac{\partial}{\partial u}D(u,v),$$

$\frac{\partial}{\partial v}f(u,v)$ is similar. Note that the derivatives of the displacement function are a scaled version of subpatch derivatives:

$$\frac{\partial}{\partial u}D(u,v) = \mathit{tileSize} \cdot \frac{\partial}{\partial \hat{u}}\hat{D}(\hat{u},\hat{v}).$$

Further, $\frac{\partial}{\partial u}s(u,v)$ can be directly obtained from the base surface. To find the derivative of $N_s(u,v)$, we note that the derivatives of the (unnormalized) normal $N_s^*(u,v)$ are found using the Weingarten equation [Do Carmo 1976] (E, F, G and e, f, g are the coefficients of the first and second fundamental form):

$$\frac{\partial}{\partial u}N_s^*(u,v) = \frac{\partial}{\partial u}s(u,v)\frac{fF - eG}{EG - F^2} + \frac{\partial}{\partial v}s(u,v)\frac{eF - fE}{EG - F^2},$$

$\frac{\partial}{\partial v}N_s^*(u,v)$ is similar. From this, we find the derivative of the normalized normal:

$$\frac{\partial}{\partial u}N_s(u,v) = \frac{\frac{\partial}{\partial u}N_s^*(u,v) - N_s(u,v)(\frac{\partial}{\partial u}N_s^*(u,v) \cdot N_s(u,v))}{\|N_s^*(u,v)\|_2},$$

$\frac{\partial}{\partial v}N_s(u,v)$ is similar. Finally, we compute $\frac{\partial}{\partial u}f(u,v)$ (analogously $\frac{\partial}{\partial v}f(u,v)$) and thus the displaced surface normal $N_f(u,v)$.

5.2 Approximate Shading

Since the computation of the derivatives of the base surface normal using the Weingarten equation is relatively costly, it is possible to approximate the computation of the displaced surface normal $N_f(u,v)$. Blinn [1978] suggests ignoring the Weingarten term, resulting in the approximate partial derivative:

$$\frac{\partial}{\partial u}f(u,v) \approx \frac{\partial}{\partial u}s(u,v) + N_s(u,v)\frac{\partial}{\partial u}D(u,v).$$

This is reasonable when the displacements are small since the term $\frac{\partial}{\partial u}N_s(u,v)D(u,v)$ becomes negligible. $\frac{\partial}{\partial v}f(u,v)$ can be approximated the same way. We discuss this further in Section 7, and quantify the performance of approximate versus accurate shading.

5.3 Rendering using Hardware Tessellation

We evaluate the base surface $s(u,v)$, its derivatives $\frac{\partial}{\partial u}s(u,v)$, $\frac{\partial}{\partial v}s(u,v)$ and the displacement function $D(u,v)$ in the domain shader. Additionally, the derivatives of the normal $\frac{\partial}{\partial u}N_s(u,v)$, $\frac{\partial}{\partial v}N_s(u,v)$ can be evaluated. These results are used to form the vertices of the triangle mesh generated by the tessellator.

The vertex attributes computed in the domain shader are then interpolated by hardware and available in the pixel shader. In the pixel shader, we evaluate the derivatives of the displacement function $\frac{\partial}{\partial u}D(u,v)$ and $\frac{\partial}{\partial v}D(u,v)$. This allows us to compute the derivatives of the displaced surface normal $\frac{\partial}{\partial u}f(u,v)$, $\frac{\partial}{\partial v}f(u,v)$ at each pixel. Therefore, we obtain $N_f(u,v)$ at each pixel that corresponds to the displaced surface. Evaluating the surface normal $N_f(u,v)$ on a per vertex basis would degrade rendering quality, due to interpolation artifacts.

As a base surface we tested Stam [1998] evaluation and feature adaptive subdivision [Nießner et al. 2012]. Both schemes work well; however, [Nießner et al. 2012] is faster, especially for high levels of tessellation. We therefore discuss this case in more detail.

5.4 Base Surface Evaluation

Feature adaptive subdivision [Nießner et al. 2012] leverages the nested polynomial patch structure of Catmull-Clark subdivision. Regular regions of a control mesh define bicubic B-spline patches that can be rendered by the hardware tessellator. Further subdivision is only needed near extraordinary vertices (a type of feature), to generate more regular regions and more patches. The limit surface will contain an infinite number of smaller and smaller patches around extraordinary vertices. However, after only a few subdivision levels, these patches will only cover a few pixels. At that point, adaptive subdivision can stop, and final patches are rendered as quads. Subdivision is carried out by GPGPU compute kernels driven by pre-computed index buffers. At each level of subdivision, regular patches corresponding to that level are generated. Additionally, the final extraordinary vertex hole filling quads are generated for rendering in a separate final pass. The advantage of feature adaptive subdivision is that the number of subdivision operations grows linearly with respect to subdivision level, rather than exponentially as it does when refining the entire mesh at each level. The high compute-to-memory bandwidth ratio of modern GPUs is exploited since evaluating bicubic B-splines using hardware tessellation performs better than streaming refined mesh vertices to and from GPU memory.

Applying the displacement function for level 0 patches is trivial since tiles correspond to patches. At higher subdivision levels, however, a patch will correspond to a subdomain of a base patch. Additionally, feature adaptive subdivision may rotate patches (by $j\frac{\pi}{2}$) to reduce combinatorics. So for each patch we store a local offset within a tile and its rotation j. The size of a patch domain is provided by its subdivision level. From these we can selectively access displacements belonging to a subpatch. After taking patch subdomains and rotations into account, all patches are rendered uniformly as described in Section 5.3. In the end we use the following control structure to access displacement data (we only need 16 bytes per patch compared to the 32 byte required by traditional texturing):

```
struct {
  ushort[2] globalTextureOffsetXY;
  ushort[2] localDomainOffsetXY;
  ushort tileSize;
  ushort rotation;
  uchar[4] tileSizeDifferences;
};
```

Rendering the extraordinary vertex quads (i.e., faces at the finest subdivision level that are not being further tessellated) requires a separate rendering pass. In this case, an exception to the rendering rule for regular patches occurs. Due to the singularity in the subdivision surface parameterization at extraordinary vertices, the directions $\frac{\partial}{\partial u}s$ and $\frac{\partial}{\partial v}s$ are not consistently defined (as they are at all other points of the surface). However, the base surface limit normal N_s is well defined at extraordinary vertices. So to obtain a consistent normal over these final quads (in particular along quad edges), we bilinearly blend between the limit normal N_s, and the displaced surface normal N_f involving the interpolated partials $\frac{\partial}{\partial u}s \times \frac{\partial}{\partial v}s$. The blending function equals 1 at extraordinary vertices (for N_s) and 0 at other quad corners (for N_f), and is performed on a per fragment basis without requiring the evaluation of the Weingarten term at the extraordinary point. Fortunately, since the quads incident on extraordinary vertices are rendered separately, there are no measurable extra costs for this special treatment. Also note that the partial derivatives of the displacement function $D(u,v)$ are restricted to be 0 at extraordinary vertices (see Section 4.2); thus, the resulting surface will be C^1 and correspond to the displacement data.

6. LEVEL-OF-DETAIL

Hardware tessellation is controlled by user specified tessellation (tess) factors assigned per patch in a hull shader program. The fixed function *tessellator unit* then generates an appropriate sampling pattern to match these inputs. This is particularly effective in the context of displacement mapping since detail can be added and removed at runtime. However, under-sampling occurs when the resolution of the sampling pattern is insufficient to reconstruct the high frequency displacement detail. This can lead to swimming artifacts since minor tess factor changes can cause significant changes in the resulting surface. Our solution is to select mip levels based on the tessellation density in order to avoid under-sampling artifacts.

6.1 Tessellation Factor Estimation

We first estimate tess factors; this includes two interior tess factors per patch, as well as a tess factor for edge patch. Adjacent patches must have the same tess factors assigned along shared edges in order to guarantee crack-free rendering. Our approach is to determine a tessellation density value for each vertex of the base mesh in a compute shader kernel, and then to propagate these values to the (sub) patches corners using bilinear subdivision.

We determine tess factors TF for base mesh vertices v (with edges e) using one of these simple methods (c is a user defined constant):

—Distance based: $TF = c \cdot \|eye - v\|_2$

—Screen space area based: $TF = c \cdot \sqrt{\sum e_i \times e_{i+1}}$

—Screen space edge length based: $TF = c \cdot \max_i \|e_i\|_2$

More elaborated methods that take surface curvature into account are also possible, but would require greater computational effort.

Once tess factors have been computed for patch corners, we assign patch edge tess factors as the maximum of the two incident corner tess factors. We treat inner tess factors analogously (maximum of opposite edge tess factors in u and v directions).

6.2 Mip Level Selection

The tess factors computed for each of the four patch corners are passed to the domain shader and are bilinearly interpolated producing the function $TF(u,v)$. Based on this interpolant we select the two adjacent mip levels $\lfloor \log 2(TF(u,v)) \rfloor$ and $\lceil \log 2(TF(u,v)) \rceil$ and linearly interpolate the resulting displacements (including the derivatives). We must also clamp this value to the maximum mip level determined in Section 4.4 in order to provide consistent results at shared boundaries where tiles with distinct resolutions meet. This allows us to guarantee a specific sampling rate of the displacement map in order to avoid under-sampling.

7. RESULTS

Our implementation uses DirectX 11 with shader code written in HLSL. Timing measurements were made on a NVIDIA GeForce GTX 480 and are provided in milliseconds.

In order to test our method we extracted displacement values from two representative models, the Dragon Head and the Monster Frog. Figure 1 shows the rendering with, and without, displacements including the control mesh of the base mesh. For these images the approximate variant (see Section 5.2) without the Weingarten term is used and the level-of-detail computation (see Section 6) is omitted. Displacements are stored using 16 bit floating points and a tile size of 16×16 (overall 2.7 MB) and 8×8 (overall 413 KB), respectively. Tessellation factors are determined adaptively based on the camera distance (637K and 328K triangles are generated) so that the models are rendered with our algorithm in 1.7 ms and 1.3 ms. Note that we also tried 32 bit floating point displacement values; both performance and visual quality changes were insignificant.

Figure 5 depicts the level-of-detail scheme proposed in Section 6 (again the Weingarten term is ignored). By blending between adjacent mip levels we achieve smooth level-of-detail transitions. We found that using Harr wavelets provided smoother results, whereas biquadratic B-wavelets preserved more detail in down sampled mip levels. Results shown here and in the video use the Haar wavelet mip level variant. Detail is dynamically added with an increased tessellation rate and vice versa. This enforces a certain sampling rate of the displacement values and thus prevents under-sampling artifacts. For these images (from left to right) tess factors 2, 4, 8, 16 are chosen resulting in rendertimes 0.6 ms, 1.0 ms, 1.7 ms, 3.3 ms, respectively.

The difference between accurate and approximate normal computation (with and without the Weingarten term) is shown in Figure 6 for the Dragon Head and Monster Frog. Patches with large displacement offsets and high curvatures show the most difference; elsewhere the results are visually indistinguishable. For these render settings (same as in Figure 1), taking the Weingarten term into account increases the rendertime from 1.7 ms to 2.6 ms and from 1.3 ms to 1.9 ms, respectively. Considering the minor shading improvements but the major performance decrease the accurate variant seems to be less appropriate for real-time entertainment applications.

Figure 7 shows performance results of our method with various setups using different tess factors. Rendering the basic version of our method (i.e., without level-of-detail and without Weingarten term) is only slightly slower than rendering the base surface without displacements at all. Both level-of-detail and accurate normal computation come at some costs, whereas taking the Weingarten term into account affects performance more drastically. We also compare our method against traditional displacement mapping combined with tangent space normal mapping. While the resulting surfaces of our method (basic version) and the traditional approach are about the same, our method achieves higher frame rates. This is attributed to the fact that we only require a single displacement texture rather than a separate displacement and normal map. Also note that tangent space normal mapping has issues with texture seams,

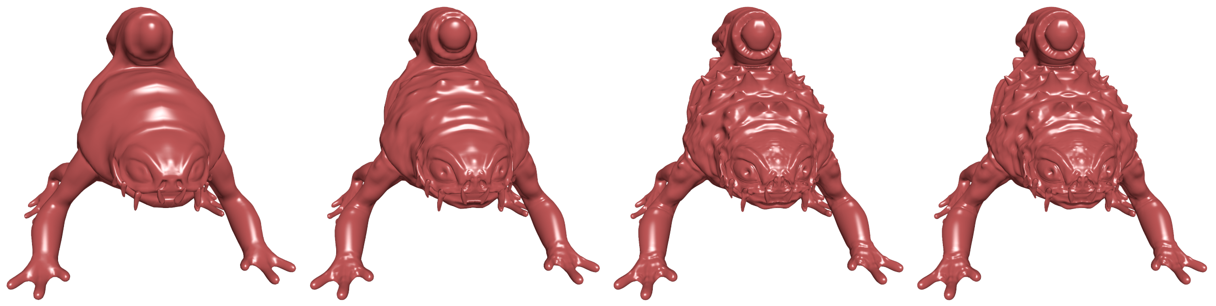


Fig. 5. Our level-of-detail scheme; pre-computed mip levels are selected based on the tess factors. These images (from left to right) are rendered using tess factors 2, 4, 8, 16 resulting in rendertimes 0.6 ms, 1.0 ms, 1.7 ms, 3.3 ms. Also note that we are linearly blending between two adjacent mip levels in order to obtain a smooth transition between selected levels.

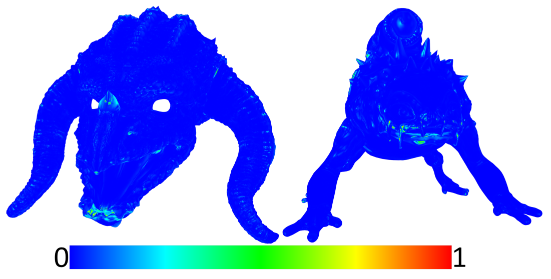
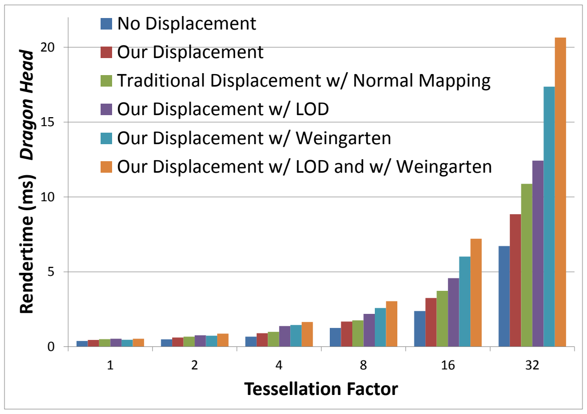


Fig. 6. Difference (Euclidean color distance in HSV) between the approximate normal computation and the accurate variant that takes the Weingarten term into account. Patches with large displacement offsets and high curvature are most different. For this setting (same as in Figure 1) the accurate variant is 46% and 53%, respectively, slower than the approximation.

under-sampling artifacts, and does not support dynamic displacements.

8. CONCLUSION

We have presented a method for rendering displaced Catmull-Clark subdivision surfaces that avoids typical artifacts that have limited their use on GPUs with hardware tessellation. These include texture seams, the need for normal maps to provide appropriate high frequency shading, and under-sampling that causes swimming. We introduced a tile based texture format for the GPU and defined an analytically smooth displacement surface using a biquadratic displacement function. Data for this function can be obtained by traditional displacement extraction (e.g., MudBox or ZBrush) or direct authoring as demonstrated in the accompanied video. We believe these advances will be useful in both the context of authoring, as well as in the runtime engine, where our method provides highly accurate shading of detailed models at high frame rates.



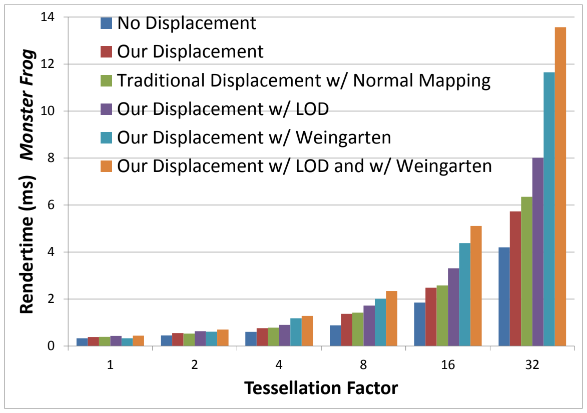


Fig. 7. Performance results for rendering the base surface, rendering the displaced surface using our method with various setups, and a comparison against traditional displacements combined with tangent space normal mapping. Note that traditional displacement mapping is always slower than our basic method even so dynamic tangent and bitangent computation (required for animation) is omitted.

ACKNOWLEDGMENTS

We would like to thank Bay Raitt of Valve Software for the Monster Frog model. Further, we would like to thank the Blender Foundation for the Dragon Head that we took from the DVD of the open movie Sintel (www.sintel.org).

REFERENCES

BERTRAM, M. 2004. Lifting biorthogonal b-spline wavelets. *Geometric Modeling for Scientific Visualization*, 153.

BLINN, J. 1978. Simulation of wrinkled surfaces. In *ACM SIGGRAPH Computer Graphics*. Vol. 12. ACM, 286–292.

BUNNELL, M. 2005. Adaptive Tessellation of Subdivision Surfaces with Displacement Mapping. *GPU Gems 2*, 109–122.

BURLEY, B. AND LACEWELL, D. 2008. Ptex: Per-Face Texture Mapping for Production Rendering. In *Computer Graphics Forum*. Vol. 27. Wiley Online Library, 1155–1164.

CATMULL, E. AND CLARK, J. 1978. Recursively generated B-spline surfaces on arbitrary topological meshes. *Computer-aided design 10,* 6, 350–355.

COOK, R. 1984. Shade trees. In *Computer Graphics (SIGGRAPH 84 Proceedings)*.

DO CARMO, M. 1976. *Differential geometry of curves and surfaces*. Vol. 1. Prentice-Hall.

DOO, D. 1978. A Subdivision Algorithm for Smoothing Down Irregularly Shaped Polyhedrons. In *Proceedings on Interactive Techniques in Computer Aidied Design, Bologna, Italy*. IEEE, 157–165.

GONZÁLEZ, F. AND PATOW, G. 2009. Continuity mapping for multi-chart textures. In *ACM Transactions on Graphics (TOG)*. Vol. 28. ACM, 109.

GUSKOV, I., VIDIMČE, K., SWELDENS, W., AND SCHRÖDER, P. 2000. Normal meshes. In *SIGGRAPH Proceedings*. ACM Press/Addison-Wesley Publishing Co., 95–102.

HALSTEAD, M., KASS, M., AND DEROSE, T. 1993. Efficient, fair interpolation using Catmull-Clark surfaces. In *Proceedings of the 20th annual conference on Computer graphics and interactive techniques*. ACM, 35–44.

LEE, A., MORETON, H., AND HOPPE, H. 2000. Displaced subdivision surfaces. In *SIGGRAPH Proceedings*. ACM Press/Addison-Wesley Publishing Co., 85–94.

LOOP, C. 1987. Smooth subdivision surfaces based on triangles. M.S. thesis, University of Utah.

LOOP, C., SCHAEFER, S., NI, T., AND CASTANO, I. 2009. Approximating subdivision surfaces with gregory patches for hardware tessellation. In *ACM Transactions on Graphics (TOG)*. Vol. 28. ACM, 151.

MYLES, A., NI, T., AND PETERS, J. 2008. Fast parallel construction of smooth surfaces from meshes with tri/quad/pent facets. *Computer Graphics Forum 27,* 5, 1365–1372.

NIESSNER, M., LOOP, C., MEYER, M., AND DEROSE, T. 2012. Feature-adaptive GPU rendering of Catmull-Clark subdivision surfaces. *ACM Transactions on Graphics (TOG) 31,* 1, 6.

PATNEY, A., EBEIDA, M., AND OWENS, J. 2009. Parallel view-dependent tessellation of Catmull-Clark subdivision surfaces. In *Proceedings of the Conference on High Performance Graphics 2009*. ACM, 99–108.

PURNOMO, B., COHEN, J., AND KUMAR, S. 2004. Seamless texture atlases. In *Proceedings of the 2004 Eurographics/ACM SIGGRAPH symposium on Geometry processing*. ACM, 65–74.

RAY, N., NIVOLIERS, V., LEFEBVRE, S., AND LÉVY, B. 2010. Invisible Seams. In *Computer Graphics Forum*. Vol. 29. Wiley Online Library, 1489–1496.

REIF, U. 1997. A refineable space of smooth spline surfaces of arbitrary topological genus. *Journal of Approximation Theory 90,* 2, 174–199.

SANDER, P., WOOD, Z., GORTLER, S., SNYDER, J., AND HOPPE, H. 2003. Multi-chart geometry images. In *Proceedings of the 2003 Eurographics/ACM SIGGRAPH symposium on Geometry processing*. 146–155.

SCHÄFER, H., PRUS, M., MEYER, Q., SÜSSMUTH, J., AND STAMMINGER, M. 2012. Multiresolution Attributes for Tessellated Meshes.

SHIUE, L., JONES, I., AND PETERS, J. 2005. A realtime GPU subdivision kernel. *ACM Transactions on Graphics (TOG) 24,* 3, 1010–1015.

STAM, J. 1998. Exact evaluation of Catmull-Clark subdivision surfaces at arbitrary parameter values. In *Proceedings of the 25th annual conference on Computer graphics and interactive techniques*. ACM, 395–404.

SZIRMAY-KALOS, L. AND UMENHOFFER, T. 2008. Displacement Mapping on the GPU-State of the Art. In *Computer Graphics Forum*. Vol. 27. Wiley Online Library, 1567–1592.

TATARCHUK, N., BARCZAK, J., AND BILODEAU, B. 2010. Programming for real-time tessellation on gpu. *AMD whitepaper 5*.

WILLIAMS, L. 1983. Pyramidal parametrics. In *Computer Graphics (SIGGRAPH 83 Proceedings)*.

YUKSEL, C., KEYSER, J., AND HOUSE, D. 2010. Mesh colors. *ACM Transactions on Graphics (TOG) 29,* 2, 15.

Received September 2008; accepted March 2009