Multiresolution Attributes for Tessellated Meshes

Henry Schäfer Magdalena Prus Quirin Meyer Jochen Süßmuth Marc Stamminger
University of Erlangen-Nuremberg

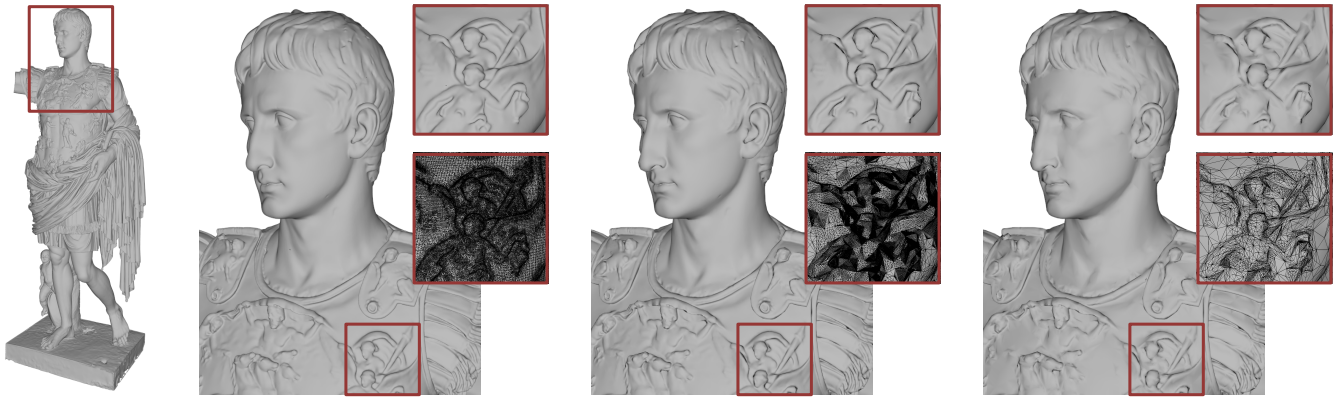

Figure 1: *Reproducing fine surface detail using our method for signal optimal displacement mapping. From left to right: Augustus model and close- ups of the full resolution model (366 MB GPU memory, 19.3 ms rendering time), and our reconstructions with fitting errors $\varepsilon_{\max} = 0.1$ (90 MB, 11.9 ms) and $\varepsilon_{\max} = 0.5$ (28 MB, 3.7 ms), respectively.*

Abstract

We present a novel representation for storing sub-triangle signals, such as colors, normals, or displacements directly with the triangle mesh. Signal samples are stored as guided by hardware-tessellation patterns. Thus, we can directly render from our representation by assigning signal samples to attributes of vertices generated by the hardware tessellator.

Contrary to texture mapping, our approach does not require any atlas generation, chartification, or uv-unwrapping. Thus, it does not suffer from texture-related artifacts, such as discontinuities across chart boundaries or distortion. Moreover, our approach allows specifying the optimal sampling rate adaptively on a per triangle basis, resulting in significant memory savings for most signal types.

We propose a signal optimal approach for converting arbitrary signals, including existing assets with textures or mesh colors, into our representation. Further, we provide efficient algorithms for mip-mapping, bi- and tri-linear interpolation directly in our representation. Our approach is optimally suited for displacement mapping: it automatically generates crack-free, view-dependent displacement mapped models enabling continuous level-of-detail.

CR Categories: I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism—Color, shading, shadowing, and texture; I.3.6 [Computer Graphics]: Methodology and Techniques—Graphics data structures and data types;

Keywords: signal dependent storage, tessellation, displacement mapping

1 Introduction

The latest feature of graphics hardware is programmable tessellation. Input geometry can be tessellated using a set of new shaders that control tessellation depth and compute the new vertex positions. Hardware tessellation can be used to amplify geometry in order to generate smooth curved surfaces or for displacement mapping.

One issue with hardware tessellation is the question how to determine the attributes for the newly generated vertices (vertex position and normal, texture coordinates, color, displacement etc.). Depending on the attribute, three possibilities are used in practice: First, attributes can simply be interpolated (e.g., texture coordinates). Second, some attributes can be determined procedurally from the barycentric coordinates (vertex positions and normals on parametric surface). Third, the attribute can be baked into a texture. In this case, the attribute is typically not fetched per vertex and then interpolated by the rasterizer, but it is directly read from the texture in a pixel shader. An important exception are displacement maps, from which values must be fetched per vertex.

Storing the attributes in textures has a number of disadvantages: first, a parameterization for the objects is needed, which requires segmentation, unwrapping the segments, and puzzling the unwrapped segments to a texture atlas. Second, it is difficult to adapt the texture resolution to the given signal, i.e., often large parts of a texture contain only little information. Finally, displacement mapping exhibits artifacts: when displacement values are read per vertex from a texture they are interpolated. Undersampling results in well visible artifacts at the silhouette of objects, and when tessellation factors are changing smoothly, the resulting mesh shows *swimming* artifacts. Mip-mapping does not solve this problem well. Another problem appears in connection with texture atlases: along texture seams slight discontinuities may appear, which are hardly visible for color textures, but result in cracks when using displacement maps.

In this paper, we propose an alternative approach to generate attributes for tessellated meshes. We store for each triangle the at-

tribute values for the vertices generated by different tessellation factors in a simple list. Whenever the triangle is tessellated at runtime, we directly fetch the optimized attribute values. Additionally, we show how attribute values are interpolated and filtered.

We further show how attribute values are determined using a multiresolution fitting procedure. The resulting fitting error is used to control tessellation factors in a level-of-detail fashion. Furthermore, for every triangle we adapt the maximum tessellation factor for which attribute values are stored. We thus easily adapt the amount of stored data to the required detail of the signal.

Our approach can be used in any rendering application using tessellation, for example for the rendering of animated subdivision surfaces. It assigns attributes to newly generated vertices with high performance. Data resolution is signal-optimized, and the signal is stored and reconstructed smoothly at multiple resolutions. Our method requires no parameterization and is fully automatic.

A particular useful application is to replace a detailed mesh by a coarse mesh and a displacement map, as shown in Figure 1. Besides the dramatically reduced storage requirements for the index buffer, we save a significant amount of memory by storing displacement samples only where needed. At the same time we can control the well-known undersampling and swimming artifacts, and completely avoid cracks along texture seams. By storing the attribute values directly in lists without making a detour via a texture, we reduce the number of stored samples and we assign attribute values directly at the position they were optimized for.

2 Related Work

The ability to tessellate a coarse input mesh to generate massive amount of geometry directly in the graphics pipeline is typically used to generate smooth surfaces via subdivision methods [Nießner et al. 2012; Loop and Schaefer 2008; Loop et al. 2009] and to add geometric detail via displacement mapping [Valdetaro et al. 2010; Tatarchuk et al. 2010]. Since all primitives in the input mesh are processed independently it is important to create consistent transitions to prevent discontinuities, especially if different tessellation factors are used between adjacent patches.

For displacement, problems appear when a texture atlas is used and interpolated texture values differ slightly, either for displacement normals or the displacement values (or both), resulting in shading discontinuities or well perceivable cracks. Much research was spent on this issue recently, either by precomputing transitions to stitch discontinuities [González and Patow 2009; Castaño 2008; Sander et al. 2003] or by creating a parametrization with consistent chart boundaries and orientation [Ray et al. 2011]; see Ni et al. [2009] for a survey. Some methods prevent visible cracks by inserting triangle strips to fill holes [Schwarz et al. 2006] or by creating a consistent edge subdivison via incremental bisection of faces with a lower subdivision [Pakdel and Samavati 2004].

Other approaches avoid texture atlas problems using alternative signal representations. Ptex [Burley and Lacewell 2008] uses per face texturing by taking advantage of implicit parametrization of quad patches. Mesh Colors [Yuksel et al. 2010] virtually subdivide the mesh and use piecewise linear interpolation similar to vertex colors to reconstruct color signals. Kavan et al. [2011] directly employs vertex colors by computing optimal attributes at vertices to resemble the input signal. Unfortunately, a pure vertex based representation is suited for low frequency signals only, since sub-triangle detail cannot be represented that way. The hybrid signal adaptive method proposed by Schäfer et al. [2012] solves this problem by combining vertex- and texture-based storage. They use optimal vertex colors for interpolation by default and textures only on triangles where vertex colors cannot reconstruct the input signal. Finally, they smoothly blend between both representations.

3 Overview

In the following, we assume that a coarse base mesh is given that is tessellated on-the-fly using hardware tessellation. One or several signals are defined on this mesh, e.g., colors, displacements, normals, or lighting data. The signals are defined on a finely tessellated reference mesh, in a texture, or procedurally (e.g., lighting or displacement of parametric surfaces). We propose to bake the signal(s) into vertex attributes for the tessellated mesh, ideally at multiple resolutions.

In order to avoid textures, which suffer from the previously mentioned shortcomings, we store the attribute values for vertices generated by tessellation in lists with each triangle. We do this for a number of different tessellation factors, resulting in *multiresolution attributes* (Section 4). Then, we perform a global fitting step to compute attribute values that optimally resemble the given signal for different tessellation factors. By controlling tessellation factors per face using a fitting error, we obtain a signal-adaptive representation (Section 5). At render time, the attribute values for all newly generated vertices are fetched – or interpolated – from our data structure. We describe how the values can be extracted efficiently from our data structure and give details on filtering and mip-mapping (Section 6).

4 Data Layout

Similarly to mesh colors [Yuksel et al. 2010], we store our data directly with the mesh. However, our subdivision scheme is guided by the standardized pattern of the hardware tessellator, i.e., we store all vertex attributes, that correspond exactly to the vertices generated by the tessellation unit, in a simple array.

To support adaptive tessellation, we use multiple attribute sets for different tessellation factors. For example, we store the signal for each power of two tesselation factor. Vertex attributes are stored separately for vertices corresponding to corner, edge, and inner vertices as depicted in Figure 2. Vertex attributes of shared edges and corner vertices are stored only once resulting in a continuous signal as long as the tessellation is consistent.

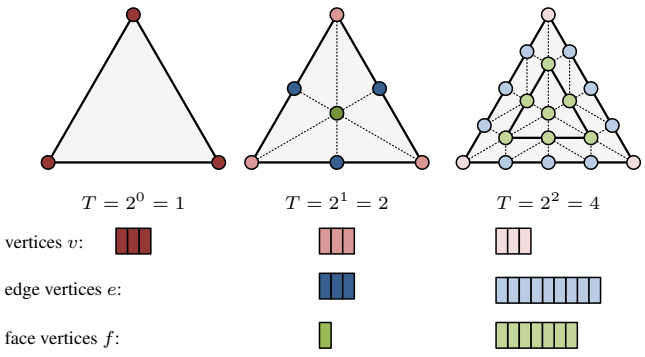

Figure 2: *Separate data storage for corner, edge, and face vertices in their corresponding arrays v, e, and f shown for different tessellation factors.*

We store all attribute data in a global array and store for each face the indices to the first attribute values of the face's corner, edges, and face vertices. For multiresolution attributes we make sure to store data corresponding to one vertex, edge, or face for all of its

levels one after another. This allows us to use only one index per list and to compute the offset to the next level on-the-fly. Additionally, we store for each vertex an error measuring the maximal signal data error occurring at a particular level.

After a face has been tessellated using a tessellation factor for which we have an attribute array, we can simply copy our data to the attributes of the vertices generated by the tessellator, and the interpolation will automatically be performed by the graphics hardware. For intermediate tessellation factors we use filtering techniques that will be described in Section 6.

5 Fitting

In a preprocessing step, we convert all signals, e.g., normal, colors, and displacement maps, that are attached to the base mesh into a representation such that the signal can later be reconstructed as vertex attributes of the tessellated base mesh. Thereby, we determine for each triangle the exact tessellation factor that is necessary to accurately reproduce the input signal. This results in a *signal optimal* representation, as within each triangle, we only store as much data as needed in order to reconstruct the input signal up to a given accuracy.

First, we convert the input signal into a discrete set of sample points that lie on the base mesh as proposed by Schäfer et al. [2012]: For each sample $\mathbf{s}_i$ of the original signal – this may be a texel midpoint, a mesh color, a stroke in an interactive painting application or a sample taken from a continuous function – we compute the barycentric coordinates w.r.t. the triangle t that contains $\mathbf{s}_i$. Let $\alpha_i, \beta_i, \gamma_i$ be the barycentric coordinates of the sample point $\mathbf{s}_i$ w.r.t. the corner vertices $v_{i,0}, v_{i,1}, v_{i,2}$ of triangle t and $f(\mathbf{s}_i)$ the signal at $\mathbf{s}_i$.

Since we store the signal as per-vertex attributes of a (tessellated) mesh, the signal value $f'(\mathbf{s}_j)$ at $\mathbf{s}_i$ will later be a barycentric interpolation of the attribute values $f_{i,0}, f_{i,1}, f_{i,2}$ at the corner vertices $v_{i,0}, v_{i,1}, v_{i,2}$ of triangle t:

$$f'(\mathbf{s}_j) = \alpha f_{i,0} + \beta f_{i,1} + \gamma f_{i,2}. \quad (1)$$

To find the *signal optimal* vertex attribute representation of the input signal, we need to estimate the function values f_i at the mesh's vertices v_i such that the average difference between the actual signal values $f(\mathbf{s}_j)$ and the interpolated signal value $f'(\mathbf{s}_j)$ is minimized. As shown by Schäfer et al. [2012], a least-squares solution to this problem can be efficiently found by solving a sparse Laplace-like linear system.

For most signals, it will not be sufficient to store the signal only at the vertices of the base mesh, since sub-triangle signal cannot be represented this way. Therefore, we compute for each triangle t_i, whether the signal within the triangle is reconstructed accurately enough by calculating the maximum deviation ε_i between a function sample $f(\mathbf{s}_j)$ and the corresponding linear interpolation $f'(\mathbf{s}_j)$:

$$\varepsilon_i = \max \mathrm{abs}(f'(\mathbf{s}_j) - f(\mathbf{s}_j)) \quad \forall \quad s_j \in t_i. \quad (2)$$

If ε_i is smaller than a user defined threshold $\varepsilon_{\mathsf{max}}$, the signal within the triangle t_i can be adequately represented by linear interpolation and we consider triangle t_i to be *converged*.

Then, we refine all triangles that are not yet converged by increasing their tessellation factor (in practice, we increase the tessellation factor from 2^i to 2^{i+1} as this allows us to compute a mip-map representation of the signal). This refinement produces new vertices inside the triangles, and thus, allows us to represent the input signal more faithfully as vertex attributes. We then repeat the fitting process for the refined mesh.

To avoid discontinuities between triangles that require different tessellation factors, we set the tessellation factor of each edge to the maximum tessellation factor of its adjacent triangles. This ensures that the result of the tessellation is always a crack-free triangle mesh without T-junctions, thus guaranteeing that the signal is continuous along edges. When fitting the vertex attributes of the refined tessellation mesh, we need to take care that the signal within triangles that have converged at an earlier level is not changed anymore, as this could move the approximation error within these triangles above the threshold $\varepsilon_{\mathsf{max}}$ again. Therefore, we *lock* all vertices that belong to a converged triangle as shown in Figure 3. Vertices on edges of locked triangles introduced by higher tessellation factors of adjacent triangles are locked as well and the respective attributes are linearly interpolated along the edge. We handle locked vertices by treating them as fixed constraints within the system of linear equations, i.e., we remove the corresponding rows from the system matrix. This reduces memory requirements and improves fitting speed, as usually only a fraction of the mesh's vertices has to be fitted at each level.

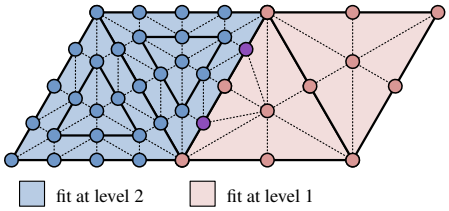

Figure 3: *Locking converged vertices during the proposed hierarchical fitting process. The pink triangles have converged at level 1. Therefore, their vertices (pink) are locked when fitting subsequent levels. The purple vertices on the edge between the pink and the light blue triangle have been inserted to produce a crack-free mesh. To ensure that the signal within the converged triangle is not changed anymore, the attribute values at the purple points are linearly interpolated between the vertex neighbors on the edge and the purple vertices are locked as well.*

The proposed fitting scheme offers two advantages: First, the signal is represented optimally, as for each triangle, we only store the amount of data that is required to reconstruct the signal within the triangle. A second advantage of our hierarchical fitting scheme is that the intermediate fitting results provide signal optimal mip-maps. When fitting a model up to the desired accuracy, we always increase the tessellation factor of non-converged triangles by a factor of 2. Thus, by storing the intermediate fitting results, we obtain a power-of-two mip-map representation that also approximates the input signal in a least-squares sense.

6 Rendering

For rendering tessellated meshes, the graphics hardware provides two programmable stages: the first one is the hull shader which is used to control tessellation by choosing an output primitive type and setting edge and face tessellation factors. Then, the mesh is tessellated and the newly generated vertices are processed by a domain shader where attributes can be assigned.

In the following we describe how attributes are assigned to tessellated meshes using our storage scheme. We start with the simplest case where tessellation factors exactly match factors for which we have data available. In the next step we show how filtering methods are applied to support intermediate tessellation factors. Finally, we show how adaptive tessellation is combined with our multiresolution storage scheme to achieve watertight meshes and smooth transitions between stored levels.

6.1 Attribute Assignment

In order to assign attributes in the domain shader, a lookup into our data structure is required. Unfortunately, the tessellation hardware does not provide a unique index for the generated vertices, so we have to derive the index from the barycentric coordinates (α, β, γ) to perform the lookup. In the following we describe the index computation for the triangle tessellation pattern, although the method can easily be extended to quads (see results).

According to the tessellation specification, triangles are first subdivided into $\lfloor T_{\text{inner}}/2 \rfloor$ concentric triangles, where T_{inner} is the inner tessellation factor set in the hull shader. Then, the outer edges are subdivided using the edge tessellation factors, whereas the edges belonging to inner concentric triangles are subdivided according to the inner tessellation factor.

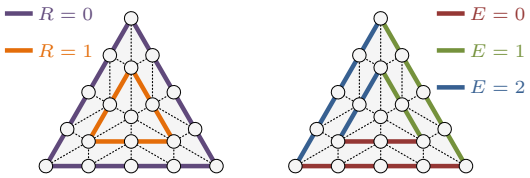

Figure 4: *Ring (left) and edge (right) indices computed from barycentric coordinates.*

This tessellation scheme allows us to compute a unique index from barycentric coordinates. For a given fractional barycentric coordinate we determine the corresponding concentric ring R, the edge E on the ring and vertex index V on the edge as shown in Figure 4 with:

$$R = \text{round}\left(\frac{3}{2} T_{\text{inner}} \cdot b_{\text{min}}\right), \tag{3}$$

where b_{min} is the minimum of (α, β, γ). The edge index E can be directly derived with:

$$E = \begin{cases} 0, \alpha = b_{\text{min}} \\ 1, \beta = b_{\text{min}} \\ 2, \gamma = b_{\text{min}} \end{cases}. \tag{4}$$

For the vertex index we project the coordinate and its corresponding edge onto the outermost triangle in a barycentric manner. The final vertex index then depends on the number of vertices on that edge. Obviously, we have to distinguish between vertices on the inner and outer edges of a triangle. Points inside the triangle always refer to the same tessellation factor T_{inner}, whereas points on the outermost concentric triangle refer to the edge tessellation factors $T^0_{\text{outer}}, T^1_{\text{outer}}, T^2_{\text{outer}}$. Thus, a general computation of the vertex index V is given by:

$$V = \frac{b'_{\mathsf{E}}}{\alpha + \beta + \gamma - 3b_{\text{min}}} (T - 2 \cdot R), \tag{5}$$

where $b' = [\beta\ \gamma\ \alpha]^T$ and

$$T = \begin{cases} T^{\mathsf{E}}_{\text{outer}}, R = 0 \\ T_{\text{inner}}, \text{otherwise.} \end{cases} \tag{6}$$

The resulting ring, edge, vertex (REV) indices together with the starting indices stored in our data structure are then used to retrieve the attribute value for the current vertex.

6.2 Filtering and Mip-Mapping

For practical applications, where arbitrary tessellation factors are desired, it is required to assign attributes at positions in between our sample positions. In the following we describe how nearest neighbor and bilinear interpolation can be implemented using our storage scheme. Let T_{current} be the currently used tessellation factors and T_{target} be the factors for which we have data available. For both interpolation schemes we use the ratio $q = T_{\text{target}}/T_{\text{current}}$ to find attributes in our data set.

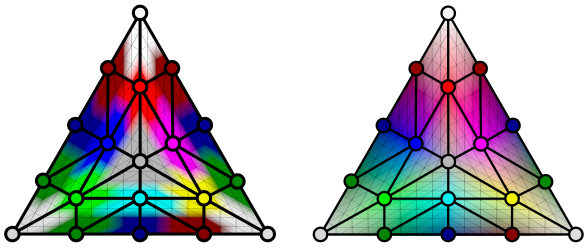

Figure 5: *Example of nearest neighbor (left) and bilinear interpolation (right) for attributes in the current tesselation level using attributes available in a different level (black edges).*

Nearest neighbor interpolation requires to find the nearest point v_{n} to the current vertex position v_{p}, for which we have data. First, we compute $REV(v_{\mathsf{p}})$ for the current vertex. The nearest neighbor in our data set can then directly be computed by rounding the indices to indices regarding the target tessellation factors:

$$\begin{aligned} R_{\mathsf{n}} &= \text{round}(q \cdot R_{\mathsf{p}}), \\ E_{\mathsf{n}} &= E_{\mathsf{p}}, \\ V_{\mathsf{n}} &= \text{round}(t \cdot V_{\mathsf{p}}), \end{aligned} \tag{7}$$

where t is the ratio of generated segments on the edge of the different tessellation levels $t = (T_{\text{target}} - 2R_{\mathsf{n}})/(T_{\text{current}} - 2R_{\mathsf{p}})$.

Bilinear filtering requires to find the four nearest neighbors to the current vertex v_{p}. Similar to nearest neighbor interpolation, we start by computing the ring, edge, vertex representation of the current vertex. But instead of rounding we compute fractional values for the ring and vertex. Then, we use simple clamping to the closest inner and outer ring and left and right vertex on these rings to determine the four neighbor indices. The final computation is then done easily with the fractional ring and vertex values as depicted in Figure 6.

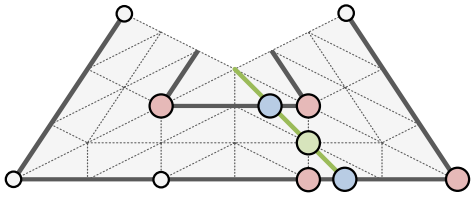

Figure 6: *Illustration of bilinear interpolation at the green vertex using data available in a coarser tessellation level (gray). The four neighboring data points (red) as well as the interpolation weights are computed by projecting the vertex to the closest inner and outer ring in the coarser representation (blue) using the ring, edge, vertex representation*

Results of both interpolation schemes are shown in Figure 5.

Mip-Mapping: To support multiple resolutions, we store optimal attribute values for all intermediate tessellation levels up to the maximum required tessellation. This is comparable to mip-mapping,

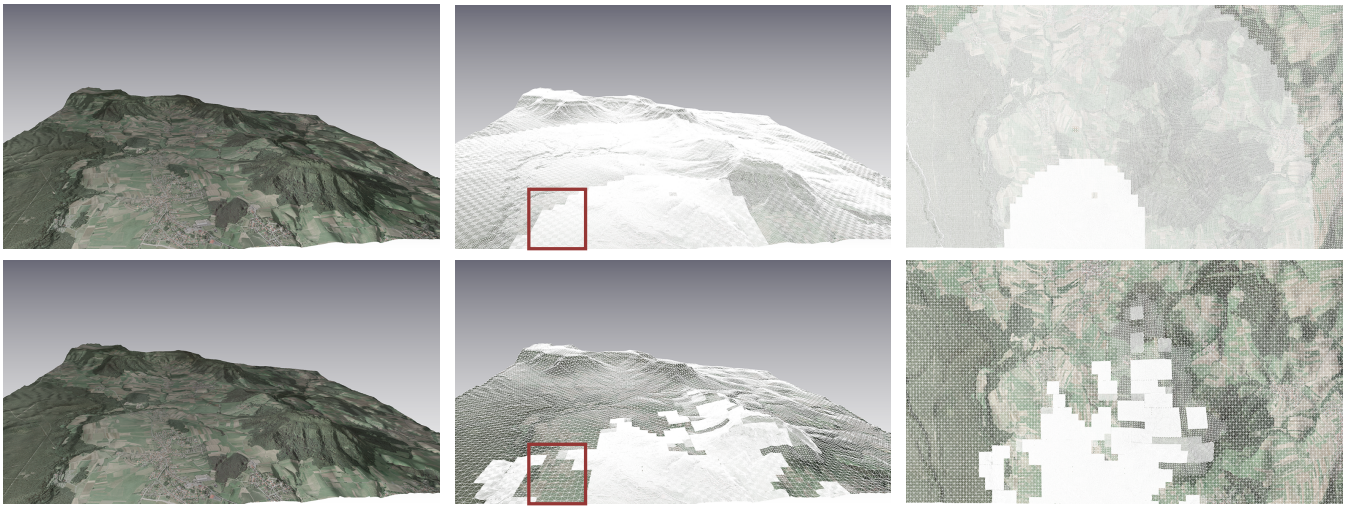

Figure 7: *Our data representation for terrain rendering using distance dependent LOD (top row) and error dependent LOD (bottom row). From left to right: standard rendering, rendering with wireframe overlay and top view for the same camera to visualize subdivision levels. The view in the top row using standard distance dependent LOD takes 4.95 ms to render, the same view in the lower row using our error dependent LOD metric takes 3.77 ms to render. The terrain data was extracted from "Geospatial Reference Data 2011" (Bavarian Administration for Surveying).*

since we store attributes for primitives generated by using power-of-two tessellation factors. But in contrast to mip-mapping, where levels are generated fine to coarse by averaging pixel values, we store optimal values generated coarse to fine. For arbitrary tessellation factors, we interpolate the stored values from the bracketing power-of-two tessellation factors, resulting in a trilinear interpolation as known from texture mip-mapping.

6.3 Signal-Adaptive Tessellation

Our signal-adaptive fitting procedure allows us to adapt tessellation factors according to the underlying signal(s).

To this end, we store with every vertex of the base mesh a list of error terms describing the signal errors of the surrounding faces using the different power-of-two tessellation levels. We set a user-defined error threshold parameter and assume that the vertex error scales with the reciprocal of the viewing distance. With a binary search on the vertex error list, we search the tessellation level that just meets the error threshold. The computed vertex levels are then used to compute edge tessellation factors (maximum level of the edge's vertices) and the inner tessellation factor (maximum level of the face's vertices).

As described above, only power-of-two tessellation factors will be chosen, resulting in popping when tessellation factors change. We can achieve a smooth transition by using fractional tessellation factors. To this end, we have to compute a fractional tessellation factor for the vertices by interpolating tessellation factors from the vertex error list according to the error threshold.

For most signals, this fractional tessellation delivers satisfying result. In the case of displacement maps, however, sampling the signal in between the original samples leads to well perceivable swimming artifacts – the resulting mesh seems to flow over the surface. This effect is also well known for displacement mapping with textures and smoothly adapted tessellation factors.

To tackle this problem, we refrain from using fractional tessellation, but always use the coarsest power-of-two tessellation factor that meets our error criterion. We avoid popping by blending the displacement with the bilinear interpolation of the next coarser level as described by Lindstrom et al. [1996]. As a result, we get a smooth transition between the two levels, but only use displacements from their original sample position. The swimming and undersampling artifacts can thus be avoided, as shown in the accompanying video.

7 Results

We successfully applied our method to quad and triangle meshes of variable sizes. All timings were measured at a screen resolution of 1280x720 using an NVIDIA GTX 480 and a Core I7 940 CPU with 2.93 GHz. The optional fitting pass applied in the preprocessing phase is currently unoptimized and takes in the order of ten minutes for the most complex mesh (100k triangles base mesh and 16M detail mesh).

The terrain shown in Figure 7 covers an area of 7.3 km^2 with a resolution of $1m^2$. We applied our method to generate displacement attributes for a 70x70 quad faces base mesh. In Figure 7, we compare standard distance-dependent LOD to our error-dependent LOD method. Choosing an error-based LOD allows us to reduce the tesselation factors and improve rendering performance without losing surface detail. Table 1 summarizes the results using our storage scheme compared to texture-based displacement mapping.

Terrain	Rendering (ms)
Nearest neighbor	2.4
Bilinear interpolation	2.7
C-LOD	3.7
Displacement from texture	3.9

Table 1: *Terrain rendering times using distance dependent LOD.*

As expected, rendering with bilinear interpolation is more expensive than nearest neighbor filtering, since additional lookups into our data structure are required. The C-LOD method requires bilinear lookups in two different resolutions and also blending, which also reduces performance. Surprisingly, in this example rendering

using our method performs better than displacement from texture. We measured a drastic increase in rendering time when displacement textures with resolutions greater than 4096^2 are used in the domain shader, since undersampling the texture results in frequent cache misses.

In Figure 8, we compare standard texture mapping with bilinear filtering to our storage scheme using nearest neighbor and bilinear interpolation. As shown in the close-up images, our method is able to reproduce the surface signal similar to standard texture filtering. Other methods that do not require an explicit parametrization could be used for such signals as well. However, a pure vertex based approach as in [Kavan et al. 2011] would not be able to represent the signal on such a coarse mesh and rendering as [Yuksel et al. 2010] requires lookups into the data buffer for each fragment, whereas our method only requires lookups at the generated vertices.

Figure 8: *Quality comparison for a lion model with color signal (from left): reference using bilinear texture lookups with wireframe overlay showing base geometry, closeup of the eye, our method with bilinear, and nearest neighbor interpolation.*

We also applied our method to the well-known Monster Frog model. Rendering displacement from texture requires 3.4 ms compared to 6.4 ms (7.2 ms for bilinear filtering) using our method. If we also store normals and color values, rendering time increases to 8.9 ms (11.4 ms). As expected, bilinear filtering reduces the performance, since additional texture lookups are necessary. Further, the performance is also influenced by our index retrieval method, which is more expensive than a simple interpolation of texture coordinates. For multi-signal rendering, a distinct index must be computed for each optimized signal separately, which explains the achieved rendering times for the Monster Frog example. Yet, if hardware would provide a unique index for generated vertices, the rendering time of our method would improve. The qualitative comparison of the Monster Frog model in Figure 9 between displacement from texture in the left image to our method on the right shows that visible artifacts may appear at chart boundaries for the displacement from texture rendering. With our method we have no such difficulties, since our edge data is always consistent, resulting in watertight meshes.

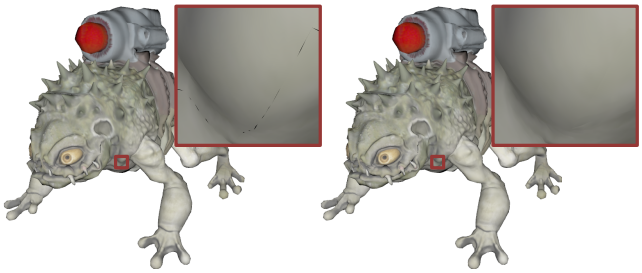

Figure 9: *Example for multi-signal: the left image shows the original Monster Frog model displaced and colored by textures, the right image shows our rendering, where displacement, normal,* and *color were stored as tessellator attributes. The left close-up shows the cracks that are common along chart boundaries when using displacement textures. Our representation (right) does not suffer from such artifacts.*

The Augustus model in Figure 1 is a scanned 3D model with 16.7M triangle faces, which was reduced to 100k faces for the base mesh. The displacement values were generated in the preprocessing step by ray casting from the base to the detail mesh. In Table 2, we compare our fitted models with different fitting errors to the reference detail mesh in terms of rendering speed and memory requirements. We clearly see that by increasing the fitting error, we can significantly reduce the amount of memory that is necessary to store data using our storage scheme. This automatically leads to better rendering performance. Note, that the memory requirement in Table 2 also includes mip-map levels.

Augustus	Rendering default (ms)	Rendering mip-map (ms)	GPU Memory (MB)
Reference	19.3	–	366
$\varepsilon_{\max} = 0.0$	49.0	85.9	307
$\varepsilon_{\max} = 0.05$	24.0	41.6	192
$\varepsilon_{\max} = 0.1$	11.9	20.9	90
$\varepsilon_{\max} = 0.5$	3.7	6.1	28

Table 2: *Memory requirements and comparison of rendering times with and without mip-mapping using different fitting errors for the Augustus model shown in Figure 1.*

8 Limitations and Future Work

Current hardware supports tessellation factors only up to 64, which limits the signal frequency that can be reproduced by our approach. This is especially the case for models with a coarse base mesh and high-frequency signal. However, performing a CPU subdivision on the mesh before our preprocessing step would solve this problem. Further, hardware support providing a unique index for each vertex generated by the tessellator would solve the costly index retrieval in the hull shader, allowing for better overall performance of our approach. At last, our data layout stores face, edge, and vertex attributes separately. This leads to incoherent memory data access and lower performance compared to the spatial locality preserved in textures. One possibility to improve the performance of our method would be to implement a more cache friendly data layout by reorganizing the storage of vertex, edge, and triangle attributes. In future we also plan to combine our displacement mapping with smooth Catmull-Clark subdivision.

9 Conclusion

We proposed a novel scheme for representing sub-triangle signals, which is optimized for meshes tessellated by graphics hardware. Storing the signal as vertex attributes of tessellated meshes offers two substantial advantages over classic textures: First, we do not require a texture atlas, which is generally difficult to compute automatically and introduces artifacts such as discontinuities between charts or problems when generating mip-maps. Second, our approach is fully signal adaptive: for each triangle we store exactly the amount of samples that is necessary to accurately reconstruct the original signal. Therefore, for a wide range of signals, our approach is also much more memory efficient than standard textures.

We proposed a fitting scheme that allows us to convert existing signals in a signal-optimal manner and showed that typical texture operations such as bilinear interpolation or mip-mapping can be performed directly on our data structure.

Finally, we showed that the proposed technique has a wide range of interesting applications, for example, level-of-detail-rendering, terrain rendering, or crack-free displacement mapping.

Acknowledgements

We would like to thank Bay Raitt of Valve Software for the Monster Frog model and the Bavarian Administration for Surveying for the terrain data. The Augustus dataset is curtesy of the Collection of Classical Antiquities (University of Erlangen-Nuremberg).

References

BURLEY, B., AND LACEWELL, D. 2008. Ptex: Per-Face Texture Mapping for Production Rendering. *Computer Graphics Forum (Proc. EGSR'08) 27*, 4, 1155–1164.

CASTAÑO, I. 2008. Next-Generation Rendering of Subdivision Surfaces. Talk at SIGGRAPH 2008.

GONZÁLEZ, F., AND PATOW, G. 2009. Continuity Mapping for Multi-Chart Textures. *ACM Trans. Graph. 28*, 109:1–109:8.

KAVAN, L., BARGTEIL, A., AND SLOAN, P.-P. 2011. Least Squares Vertex Baking. *Computer Graphics Forum (Proc. EGSR'11) 30*, 4, 1319–1326.

LINDSTROM, P., KOLLER, D., RIBARSKY, W., HODGES, L. F., FAUST, N., AND TURNER, G. A. 1996. Real-Time Continuous Level of Detail Rendering of Height Fields. In *Proceedings of SIGGRAPH'96*, 109–118.

LOOP, C., AND SCHAEFER, S. 2008. Approximating Catmull-Clark Subdivision Surfaces with Bicubic Patches. *ACM Trans. Graph. 27*, 1, 8:1–8:11.

LOOP, C., SCHAEFER, S., NI, T., AND CASTAÑO, I. 2009. Approximating Subdivision Surfaces with Gregory Patches for Hardware Tessellation. *ACM Trans. Graph. 28*, 151:1–151:9.

NI, T., CASTAÑO, I., PETERS, J., MITCHELL, J., SCHNEIDER, P., AND VERMA, V. 2009. Efficient Substitutes for Subdivision Surfaces. In *ACM SIGGRAPH 2009 Courses*, 13:1–13:107.

NIESSNER, M., LOOP, C., MEYER, M., AND ROSE, T. 2012. Feature Adaptive GPU Rendering of Catmull-Clark Subdivision Surfaces. *ACM Trans. Graph. 30*, X, accepted.

PAKDEL, H. R., AND SAMAVATI, F. 2004. Incremental Adaptive Loop Subdivision. *Computational Science and Its Applications ICCSA 2004*, 237–246.

RAY, N., NIVOLIERS, V., LEFEBVRE, S., AND LEVY, B. 2011. Invisible Seams. *Computer Graphics Forum (Proc. EGSR'10) 29*, 4, 1489–1496.

SANDER, P. V., WOOD, Z., GORTLER, S. J., SNYDER, J., AND HOPPE, H. 2003. Multi-Chart Geometry Images. In *Proceedings of SGP'07*, 146–155.

SCHÄFER, H., SÜSSMUTH, J., DENK, C., AND STAMMINGER, M. 2012. Memory Efficient Light Baking. *Computer & Graphics 36*, X.

SCHWARZ, M., STAGINSKI, M., AND STAMMINGER, M. 2006. GPU-Based Rendering of PN Triangle Meshes with Adaptive Tessellation. In *Proceedings of VMV'06*, 161–168.

TATARCHUK, N., BARCZAK, J., AND BILODEAU, B. 2010. Programming for Real-Time Tessellation on GPU.

VALDETARO, A., NUNES, G., RAPOSO, A., FEIJO, B., AND DE TOLEDO, R. 2010. LOD Terrain Rendering by Local Parallel Processing on GPU. In *Proceedings of the Brazilian Symposium on Games and Digital Entertainment 2010*, 181–188.

YUKSEL, C., KEYSER, J., AND HOUSE, D. H. 2010. Mesh Colors. *ACM Trans. Graph. 29*, 2, 15:1–15:11.