Interactive Rendering of Meso-structure Surface Details using Semi-transparent 3D Textures

Jean-François Dufort Luc Leblanc Pierre Poulin

LIGUM, Dép. I.R.O., Université de Montréal

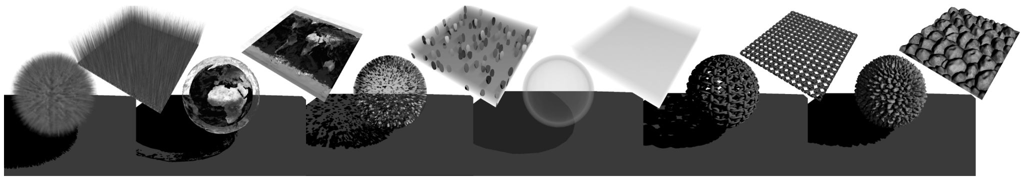


Figure 1: Triangulated sphere mapped with (from left to right) semi-transparent fur; semi-transparent clouds casting shadows on the Earth displaced with color texture; ellipsoids trapped in red semi-transparent material with volumetric shadows; uniform absorption in green medium; opaque chain model; rocks with masking and silhouettes. All shadows are cast in the volume and computed dynamically.

Abstract

Several techniques have been introduced to display meso-structure surface details to enhance the appearance of complex surfaces. One strategy is to avoid altogether costly semi-transparency in interactive contexts. However when dealing with hierarchical surface representations (important to filtering) and complex light transfers, semi-transparency must be treated. We propose a method that combines in a novel way multiple techniques from hardware rendering and volume visualization in order to render on current graphics hardware semi-transparent meso-structure surface details stored in a 3D texture. The texture is mapped to an outer shell defined by tetrahedra extruded from the surface triangular mesh. The tetrahedra are first sorted on the CPU to be rendered in correct order. The attenuated color for each ray traversing each tetrahedron is efficiently computed with a hardware voxel traversal algorithm. The resulting structure is general enough to simulate at interactive rates from semi-transparent surface details to opaque displacement maps, including several surface shading effects (visual masking, self-shadowing, absorption of light).

1 Introduction

The field of real-time rendering has greatly improved over the recent years, thanks to the phenomenal progress of graphics hardware technology. Efficient high-quality techniques are now being integrated on the GPU to produce photo-realistic images in real time or at interactive frame rates.

Surface appearance is essential to photo-realism, and the meso-structure surface details, that lay between the geometry itself and the micro-structure details (BRDF), play an important role in this appearance. This level of detail creates an intermediate representation that reduces aliasing and the number of triangles sent to the GPU. Typical meso-structure representations include 2D textures, displacement maps, and 3D textures. While being crucial to realism, 2D textures suffer from their inherent lack of depth. The silhouettes and overall appearance are greatly improved with the use of displacement maps, which increases the number of small triangles rendered. Unfortunately, the surface details must then be well represented by a height field. This is not the case for surfaces covered for instance by fur, fabrics, or trees (over hills). Even in the case of opaque surface details, correct filtering of this volumetric data for texture minification introduces semi-transparency where opaque and empty voxels are merged. Semi-transparent 3D textures, borrowed from volume rendering, alleviate many of these problems. As the surface details are treated in 3D, they can more easily simulate visual effects such as semi-transparency, self-shadowing, complex light interactions, etc. The ability to render semi-transparent textures is therefore necessary for quality images. This flexibility comes however at the expense of an increased rendering time.

We present an implementation of semi-transparent 3D texture rendering on GPU. It is simple, general, extensible, and can also handle the special case of opaque displacement mapping, although less efficiently than other current techniques. Because it relies on almost no precomputations, our method allows the animation of the underlying surface, the deformation of the outer shell containing the 3D texture, the modification of the surface parameterization, and the animation of the 3D texture. A number of surface shading effects are demonstrated to illustrate this generality. Our contributions include a combination of several hardware rendering and volume rendering techniques to render both semi-transparent and opaque volumetric textures. We provide a general framework that allows for mesh deformation and animated 3D textures. Finally, we illustrate rendering effects our technique can produce (*e.g.,* anti-aliasing, shadowing, masking), and discuss some of our experiments with current hardware that lead to some counter-intuitive realizations.

The paper is organized as follows. In Section 2, we briefly review the related work. Then, we explain our technique (Sections 3 and 4), present specific aspects of our implementation (Section 5) (*e.g.,* antialiasing, voxel traversal), demonstrate its generality and discuss some results of our technique (Section 6), and conclude (Section 7).

2 Previous Work

The techniques applied at the meso-structure level of surface detail representation can be divided in three categories: 2D textures, displacement maps, and 3D textures.

2.1 2D Textures

2D color and bump textures are essentially the most common techniques used to represent meso-structure surface details. Unfortunately they give a sense of wallpaper, as no view-dependent masking and shadowing effects can be observed (Figure 2 a). Some techniques have been introduced to improve these masking effects. Parallax mapping [25] and relief mapping [19] (Figure 2 b) are such examples. Bidirectional texture functions (BTFs) [4] capture 2D textures from different view and light directions. Rendering requires only to interpolate between textures according to current view and light directions. While providing effective solutions in real time, all these techniques still suffer from flat silhouettes and lighting effects limited to the surface.

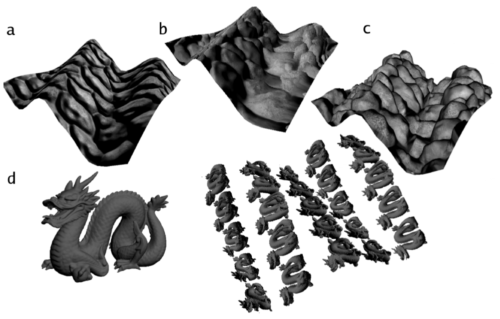


Figure 2: (a) Bump mapping and (b) relief mapping do not result in correct silhouettes as opposed to (c) displacement mapping. Our technique provides masking, correct silhouettes, and self-shadowing for geometry (d) that cannot be represented by height fields. (Color plate A) Semi-transparency adds correct blending of color and volumetric shadows.

2.2 Displacement Maps

In displacement mapping [3, 6], the geometry in each texel is moved. This results in correct view-dependent effects such as masking between surface details, details occlusion on other objects and silhouettes (Figure 2 c), but for a finely tessellated texture, can produce a large quantity of tiny triangles. A set of precomputed view-dependent displacement maps [22] can also reduce the number of geometry sent to the GPU, but requires a dense set of view directions. Other approaches consist of ray tracing inside the displacement map in a fragment program [9, 19], or image warping textures augmented with a z-component [16].

2.3 3D Textures

Ray tracing a complete 3D texture, inspired by volume rendering techniques, has been very popular to simulate surface details at various resolutions [10, 17]. It was however applied mostly for offline high-quality rendering. Meyer and Neyret [15] tile the 3D geometry with prisms and render each prism as a set of parallel textured triangles (slices). Instead of generating the slices on a per-prism basis, Lensch *et al.* [13] generate planes through the entire object in back-to-front order to handle semi-transparency. However, they can require a very large number of slicing planes in order to intersect all the details contained in the volumetric texture. Generalized displacement mapping [23] breaks away from these slicing strategies. In a pre-processing step, a 5D visibility structure is built by ray tracing the geometry from many points within the 3D texture space. While the visibility structure is compressed to about 2-4 MB for typical (64 $\times$ 64 $\times$ 32) 3D textures, it remains valid only for static 3D textures of opaque surfaces. More recently, a concurrent technique [7], similar to the one we present here, has been developed for opaque displacement maps. This technique is limited to displacement maps on locally planar surfaces. Since no integration is performed on the ray, semi-transparency is impossible. Also, a similar software based method was developed concurrently [20]. Although the two techniques are similar, the mapping between shell space and texture space is different. Also, their software implementation allows the authors to perform ray integration in object space. While providing superior image quality, their rendering times are two orders of magnitude higher.

2.4 Volume Rendering

Volume rendering algorithms (such as [26, 24, 12]) also inspired us. Projected tetrahedra [21] algorithms proceed by subdividing volumetric data into a grid of tetrahedra. Then, each tetrahedron is sent to the pipeline as a group of triangles depending on its projective profile. Colors are interpolated from each vertex using a pre-integrated color lookup table dependent on the opacity of the material within the volumetric data and the depth of each tetrahedron.

3 Meso-structure Surface Details

In this section, we describe the structure support for our meso-structure surface details. A 3D texture tiled over a triangular mesh offers a very flexible format to represent surface details.

To build the support for the 3D texture, we extrude the triangles of the surface mesh into prisms, similarly to [23]. This results in a 3D layer on top of our surface triangle mesh. Each surface vertex is extruded along the direction of its normal. Unlike [23], we furthermore subdivide each prism into three tetrahedra to ensure that each volumetric primitive is convex (Figure 3). Since our rendering technique requires no precomputations, we can modify the thickness of this layer, the extrusion direction, apply neighborhood operators to the extrusion, or animate the underlying triangle mesh. In our work, we focused our attention to the rendering of the texture and not on the modeling of this extrusion. We invite the reader to check the work of Peng *et al.* [18] for an efficient method to model such a layer on top of a surface avoiding self-intersection between prisms.

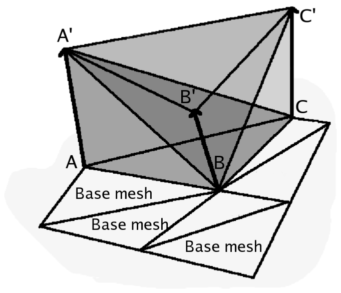


Figure 3: Each surface triangle is extruded into three tetrahedra. Tetrahedra are efficiently sorted for any given viewpoint (Section 4.1).

Each extruded vertex has a texture coordinate (u, v, w) in the 3D texture, a world position, a local frame of reference (used for shading), and additional extrusion data. This data includes the plane equation (A^o, B^o, C^o, D^o) of the supporting plane for each triangle sharing this vertex. In this paper, the superscripts ^{o} and ^{t} represent the object and texture spaces, respectively. This data also includes an affine 3×4 matrix $\mathbf{M}^{o \to t}$ that transforms a point of tetrahedron in 3D object space to the 3D texture space. All this additional data is used by the vertex shader in the rendering phase of the algorithm.

4 Rendering

The rendering procedure is composed of three steps (Figure 4):

(1) For semi-transparent 3D textures, the visible tetrahedra are identified in back-to-front order using an image-space cell sorting algorithm (SXMPVO).

(2) For each tetrahedron, we compute the entry and exit points of the viewing ray. We map these two points from object space to 3D texture space.

(3) A numerical integration for light transport is performed on this ray segment inside the 3D texture as a weighted summation along the voxel traversal.

4.1 Tetrahedra Sorting

Semi-transparent primitives must be sorted to be rendered correctly by alpha blending. Depth peeling [8] renders the tetrahedron faces in correct order, but each depth layer requires a new complete rendering pass, and the possibly high number of polygons projecting in any given pixel makes this solution impracticable. Because of the high depth complexity (around 40 triangles in one pixel), it took 20 seconds for our implementation of depth peeling to render the vase model (Color plate D) compared to 0.5 second with our CPU sorting strategy (explained next).

Another approach considers sorting the primitives before sending them to the graphics pipeline. We implemented an image-space cell sorting algorithm called SXMPVO [2]. Given a set of tetrahedra and a projection, SXMPVO returns a correct visibility ordering if there are no cycles in the set. Since by construction our tetrahedra do not intersect, configurations that could introduce cycles are rare in most well-formed meshes. Cycles are detected during the sorting and could be treated with a tetrahedra clipping strategy.

The SXMPVO algorithm proceeds in two steps. First, it sorts each pair of adjacent tetrahedra sharing one face (triangle). If by convention a triangle normal points outward its tetrahedron, the tetrahedron with the back-facing shared triangle (with respect to the viewpoint) hides its adjacent tetrahedron.

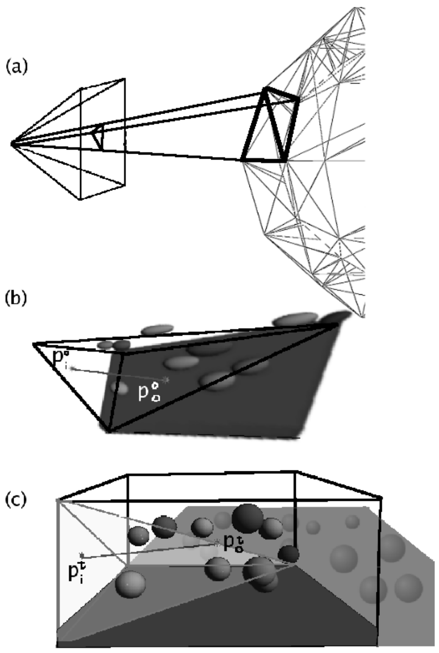


Figure 4: Overview of the rendering algorithm. (a) For semi-transparent textures, the tetrahedra are sorted by SXMPVO. A pixel covered by a tetrahedron front-facing triangle determines an entry point p_i^o. (b) The exit point p_o^o from this tetrahedron is determined by intersecting the viewing ray with the three other triangles. (c) The segment p_o^o - p_i^o is transformed in 3D texture space, where we use ray marching to accumulate its color contribution, then blended in the pixel.

Then the unshared tetrahedron faces are all projected in a simplified A-buffer. For each A-buffer pixel, the triangles are processed in back-to-front order, and the tetrahedron corresponding to a triangle is marked as behind the previous one. The remaining set of unmarked tetrahedra is a starting point for a depth-first traversal into the graph of “behind” relationships giving a correct visibility ordering of the tetrahedra.

4.2 Ray-tetrahedron Intersection

Each tetrahedron is rendered as a single primitive. We need to compute the entry and exit points of the viewing ray in the current tetrahedron, and to map these points to 3D texture coordinates (Figure 4 b and c). For efficiency reasons, only front-facing triangles are sent to the graphics pipeline. A vertex shader computes the ray on which we perform ray marching in the fragment shader. Each vertex of each front-facing triangle of the tetrahedra are sent to the pipeline. The vertex attributes are its 3D texture coordinates and the plane equation of the "opposite" triangle in the tetrahedron. This opposite triangle is the triangle of the tetrahedron formed without the current vertex. The three planes for this front-facing triangle are used to compute the exit point for the viewing ray.

In the vertex shader, the entry point in the tetrahedron of the viewing ray in object space corresponds to the current vertex position in 3D space. In order to find the exit point, we compute the intersection of the viewing ray with the three other planes supporting the three other faces of the current tetrahedron. Since the current vertex is shared by two of the three other triangles, two of the three intersections are the vertex position itself and do not need to be computed.

The corresponding 3D texture coordinates are found by multiplying the computed intersection point by the matrix $\mathbf{M}^{o \rightarrow t}$ previously defined (Section 3). Thus, the rasterizer interpolates the entry point p_i^o and the exit point p_o^o through the tetrahedron, both points in object space. The corresponding texture coordinates (u, v, w) for the two points (p_i^t and p_o^t) are also interpolated by the rasterizer. One must be careful when interpolating the intersection points because this value cannot be linearly interpolated on the surface of the triangle so a perspective correction term (in addition to the one provided by the hardware) must be used for screen interpolation. Although similar, this intersection scheme is simpler than the one found in [23] because we use convex tetrahedra instead of prisms with arbitrary shapes. Finally, the fragment shader integrates the color contribution of each voxel along the ray segment to compute its final color.

4.3 Light Transport Integration

The intersection calculation allows us to define the ray (and its counterpart V^o in object space)

$$V^t = p_i^t + \lambda(p_o^t - p_i^t)$$

where $\lambda \in [0, 1]$.

The final color returned by the fragment shader is the result of the accumulation by alpha blending of the color contribution of each voxel traversed by V^t in the 3D texture. The quality of the reconstruction is thus limited by the resolution of the volumetric texture and the sampling algorithm along the ray segment.

Depending on the sampling scheme, the distance between samples may not be uniform and certainly does not correspond to the sampling distance approximated by the alpha channel of the texture. The fragment shader must correct the opacity of a sample depending on the distance traveled by light ($\Delta t \|V^o\|$) in object space (Color plate D and G). We use the following correction term for alpha:

$$\alpha_i = 1 - (1 - \alpha_i^t)^{\Delta t \|V^o\|}$$

where α_i^t is the opacity fetched from the 3D texture at this sample location and α_i is the corrected alpha value.

Ray marching toward the light at sample points along the segment is possible. Unfortunately, at this stage, like [23], we do not have access to other tetrahedra toward the light source, and therefore neglect all light scattering inside the medium. We could use a hardware implementation of a deep shadow map [11] instead of a regular shadow map, but it would be too costly to recompute it in presence object deformations or dynamic lights.

Finally, the color returned by each visible tetrahedron is alpha blended in the current color buffer with the factors $(1 - \alpha_{dst}, 1)$.

5 Implementation

5.1 Voxel Traversal

As it will be executed for every tetrahedron viewed through each pixel, the voxel traversal for the 3D texture algorithm must be very efficient. We implemented several methods and ran some performance tests. First we tried a 3D DDA approach [1] because it guarantees that no voxels are missed. However, we found out that with high resolution 3D textures (our technique is not limited by the resolution of the textures), this algorithm results in a large number of texture fetches and thus to very poor performances. We tried adaptative sampling schemes that take into consideration the length of the ray segment inside the volumetric texture and the resolution of the texture, combined with an empty-space skipping [7] approach. The former needs a variable number of iterations in the main loop for the integration and the latter uses a large number of dependent texture lookups. Surprisingly, these enhancements to the brutal algorithm offered lower performances.

In our final implementation, we used a fixed number of samples for an image. The number of samples is a compromise between quality and speed, and can be selected at runtime. Even if we oversample often (corners of tetrahedra need less samples), this solution turned out to be the most efficient.

5.2 Antialiasing

Filtering meso-structure details has often been neglected in related work. However, lack of filtering results in very apparent under-sampling artifacts. Correct filtering of such surface details is usually very difficult to achieve because of the view dependency of the masking function within the volumetric texture. We ignore this view dependency and perform a mipmap-based filtering similar to [5].

Starting with the lowest (most detailed) level of the filtering pyramid, we combine layers of the volumetric texture by using the opacity equation [14] to filter along the z dimension (height) of the volumetric texture (top-down). We also average the color (weighted by the opacity) of the combined layers to filter in the two other (horizontal) dimensions as it is done in a regular mipmap. Despite the lack of view-dependent filtering, we obtain satisfying results that remove much of the aliasing artifacts.

Merging opaque and empty voxels in this filtering scheme introduces semi-transparency, which is correctly handled by our algorithm, as illustrated in Figure 5.

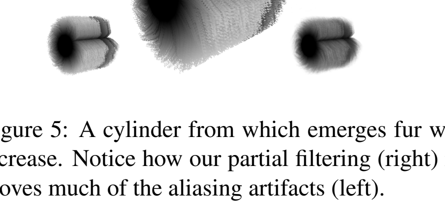


Figure 5: A cylinder from which emerges fur with a crease. Notice how our partial filtering (right) removes much of the aliasing artifacts (left).

5.3 Rendering Effects

In our implementation, each voxel in the 3D texture has an associated normal stored in another 3D texture (3D normal map). The normals can be found by the same sampling scheme used to construct the 3D texture or using the gradient of the data. In the rendering step, a local reference frame is interpolated between every vertex of the tetrahedron to perform a lighting calculation for every sample we take in the fragment shader.

To produce shadows, we use a shadow map. The world space position of the entry point of the ray segment in the tetrahedron is known, and so is the world space position of the exit point. In the ray marching step, we use the interpolated world space position as the point to query the shadow map. This scheme simulates self-shadowing of meso-structure and even volumetric shadows because each sample in the ray marching algorithm is shaded in a semi-transparent texture (Color plate E (right)). Also, the interpolation of world space position of each sample allows us to use z-buffering as a visibility algorithm to produce visual masking.

Other effects can be achieved such as light emission and absorption since we perform ray marching in volumetric texture space. The alpha channel of the 3D textures allows us to use a 3D texture in which light absorption is not uniform in resulting in regions of various opacities on the surface (Color plate D). The color channels can be used to select special voxels where light is added in the integral, resulting in a glow effect (Color plate G).

6 Results

All of our images were rendered at 512×512 from our implementation on an NVIDIA 6800 GT of 256 MB, installed on a processor Athlon64 3500+ running Linux.

Our approach has a number of advantages over other previous techniques. First, we fully support semi-transparent textures (Color plate A, D to G), which is necessary to properly filter voxels. This generality comes at the expense of a depth sorting step. This bottleneck hinders our performance, compared to other techniques limited to only opaque volumetric textures. In their situation, no attenuation calculations are required and precomputed visibility can be exploited. In our case, we can use any volumetric texture, not limited to height field based surface details. The results produce sharp silhouettes for any volumetric texture (see Color plate). Finally, since we do not rely on precomputations, both the geometrical model and the texture can be animated (Color plate B and F).

Our memory requirements are divided between the extruded mesh (the tetrahedra) and the volumetric texture. As previously mentioned (Section 3), every base mesh triangle is extruded into three tetrahedra. Thus, each vertex is duplicated and the number of triangles is multiplied by 12 (shared faces are counted twice). This part of the memory cost is comparable with other extrusion based hardware techniques such as generalized displacement maps [23] (shared faces are also counted twice). However, the memory cost for the volumetric data is smaller since we use no precomputed visibility functions. We can use textures with a resolution up to $512 \times 512 \times 32$ with no additional cost than storing the texture data ($x \times y \times z \times RGB\alpha$), plus its equivalent 3D texture for the detail normals.

Table 1 summarizes our results.

In Color plate (E left), we show an application of our technique on a highly triangulated model (8640 triangles for the base mesh, 25920 tetrahedra in total) using a texture of a tree (resolution: $128 \times 128 \times 32$). We achieve 2 fps. In order to handle curved objects, the base mesh must be tesselated. In this particular case, the texture is opaque, so no sorting is needed. To improve on the performance, we tried to use a first pass to render the base (opaque) layer without the fragment shader, and then to use the early z-discard in a subsequent pass to cull fragments that are hidden. We found that the performance gains were negligible.

The same texture was applied to a simpler base mesh, but this time, using a semi-transparent fog within the texture to add atmosphere in the scene (Color plate E (right)). The extruded mesh contains 96 tetrahedra. For each image, the tetrahedra are sorted and taking 15 samples are taken along the ray; 22% of the time is spent on the sorting and the other 88% is spent on the rendering, thus achieving 16 fps. This example shows rendering effects of our technique such as masking, absorption by the semi-transparent medium, and shadowing within the fog.

Our technique adds the possibility to use semi-transparent textures, but can reproduce results similar to the ones provided by other algorithms (Color plate C). Finally, the Color plate B and F show frames of animations (www.iro.umontreal.ca/labs/infographie/papers). They illustrate that our algorithm handles both animated textures and mesh deformations.

7 Conclusion and Future Work

We presented a technique to interactively render, on current graphics hardware, meso-structure surface details represented as semi-transparent 3D textures. Our technique makes no assumptions on the nature of the 3D texture, and is therefore applicable to animations resulting from dynamic textures, surface deformation, and texture reparameterization. Most previous techniques rely on visibility pre-processing or are limited to constrained lighting effects. Our method is general enough to support several visual and lighting effects, as demonstrated by our results.

While our method offers interactive performances for several effects, it is still too costly to be integrated in real-time systems. Expected future developments in graphics hardware, especially in 3D texturing (caching and performance), parallelism at the fragment stage, and faster depth sorting algorithms will have a direct impact on the performance of our method. Also, we believe that better visibility culling can increase the performance of our algorithm. This would spare the rendering time required to integrate invisible parts of tetrahedra reducing the impact of voxel traversal.

The concept of 3D textures to replace finer geometrical details has been around for many years [10, 17]. Beside reducing the geometry sent to the GPU, it offers a great opportunity to (pre-)filter this geometry. We hope the recent advances in rendering meso-structure surface details and image-based modeling and rendering will revive these reflections on hierarchical levels of detail, and stir graphics hardware design toward more sophisticated and integrated rendering capacities.

Acknowledgments

We acknowledge continuous financial support from NSERC. Alla Sheffer and Baining Guo graciously provided sample geometrical models and textures. Several reviewers contributed insightful comments to this final version.

Scene	Color plate	# Triangles	Texture Size	# Samples	FPS
Teapot (Beethoven)	G	25704	$128 \times 128 \times 32$	10	4
Teapot (weave)	C	25704	$64 \times 64 \times 32$	20	5
Vase (trees)	E	103680	$128 \times 128 \times 32$	10	2
Vase (dragons)	D	103680	$128 \times 128 \times 32$	10	2
Plane (trees)	E	384	$128 \times 128 \times 32$	15	16
Tube (rocks)	B	1440	$512 \times 512 \times 32$	10	30

Table 1: Summary of our results. The number of triangles is the total number sent to the pipeline: number of triangles in base mesh $\times$ 3 tetrahedra per base triangle $\times$ 4 triangles per tetrahedron.

References

[1] J. Amanatides and A. Woo. A fast voxel traversal algorithm for ray tracing. In *Eurographics '87*, pages 3–10, 1987.

[2] R. Cook, N. Max, C.T. Silva, and P.L. Williams. Image-space visibility ordering for cell projection volume rendering of unstructured data. *IEEE Trans. on Visualization and Computer Graphics*, 10(6):695–707, 2004.

[3] R.L. Cook. Shade trees. In *SIGGRAPH 1984*, pages 223–231, 1984.

[4] K.J. Dana, B.v. Ginneken, S.K. Nayar, and J.J. Koenderink. Reflectance and texture of real-world surfaces. *ACM Trans. on Graphics*, 18(1):1–34, 1999.

[5] P. Decaudin and F. Neyret. Rendering forest scenes in real-time. In *Rendering Techniques 2004*, pages 93–102, 2004.

[6] M. Doggett and J. Hirche. Adaptive view dependent tessellation of displacement maps. In *SIGGRAPH 2000*, pages 59–66, 2000.

[7] W. Donelly. *GPU Gems 2*, chapter 8. Addison Wesley, 2005.

[8] C. Everitt. Interactive order-independant transparency. NVIDIA Corp., White Paper, 1999.

[9] J. Hirche, A. Ehlert, and S. Guthe. Hardware accelerated per-pixel displacement mapping. In *Graphics Interface 2004*, pages 153–160, 2004.

[10] J.T. Kajiya and T.L. Kay. Rendering fur with three dimensional textures. In *SIGGRAPH 1989*, pages 271–280, 1989.

[11] T.-Y. Kim and U. Neumann. Opacity shadow maps. In *Eurographics Workshop on Rendering*, pages 177–182, 2001.

[12] M. Kraus, W. Qiao, and D.S. Ebert. Projecting tetrahedra without rendering artifacts. In *IEEE Visualization 2004*, pages 27–34, 2004.

[13] H.P.A. Lensch, K. Daubert, and H.-P. Seidel. Interactive semi-transparent volumetric textures. In *Vision, Modeling, and Visualization*, pages 505–512, 2002.

[14] M. Levoy. Efficient ray tracing of volume data. In *SIGGRAPH 1990*, pages 245–261, 1990.

[15] A. Meyer and F. Neyret. Interactive volumetric textures. In *Eurographics Rendering Workshop 1998*, pages 157–168, 1998.

[16] M.M. Oliveira Neto, G. Bishop, and D. McAllister. Relief texture mapping. In *SIGGRAPH 2000*, pages 359–368, 2000.

[17] F. Neyret. Modeling, animating, and rendering complex scenes using volumetric textures. *IEEE Trans. on Visualization and Computer Graphics*, 4(1):55–70, 1998.

[18] J. Peng, D. Kristjansson, and D. Zorin. Interactive modeling of topologically complex geometric detail. In *SIGGRAPH 2004*, pages 635–643, 2004.

[19] F. Policarpo, M.M. Oliveira Neto, and J. Comba. Real-time relief mapping on arbitrary polygonal surfaces. In *I3D 2005*, pages 155–162, 2005.

[20] S. D. Porumbescu, B. Budge, L. Feng, and K. I. Joy. Shell maps. In *SIGGRAPH 2005*, pages 626–633, 2005.

[21] P. Shirley and A. Tuchman. A polygonal approximation to direct scalar volume rendering. In *Workshop on Volume Visualization 1990*, pages 63–70, 1990.

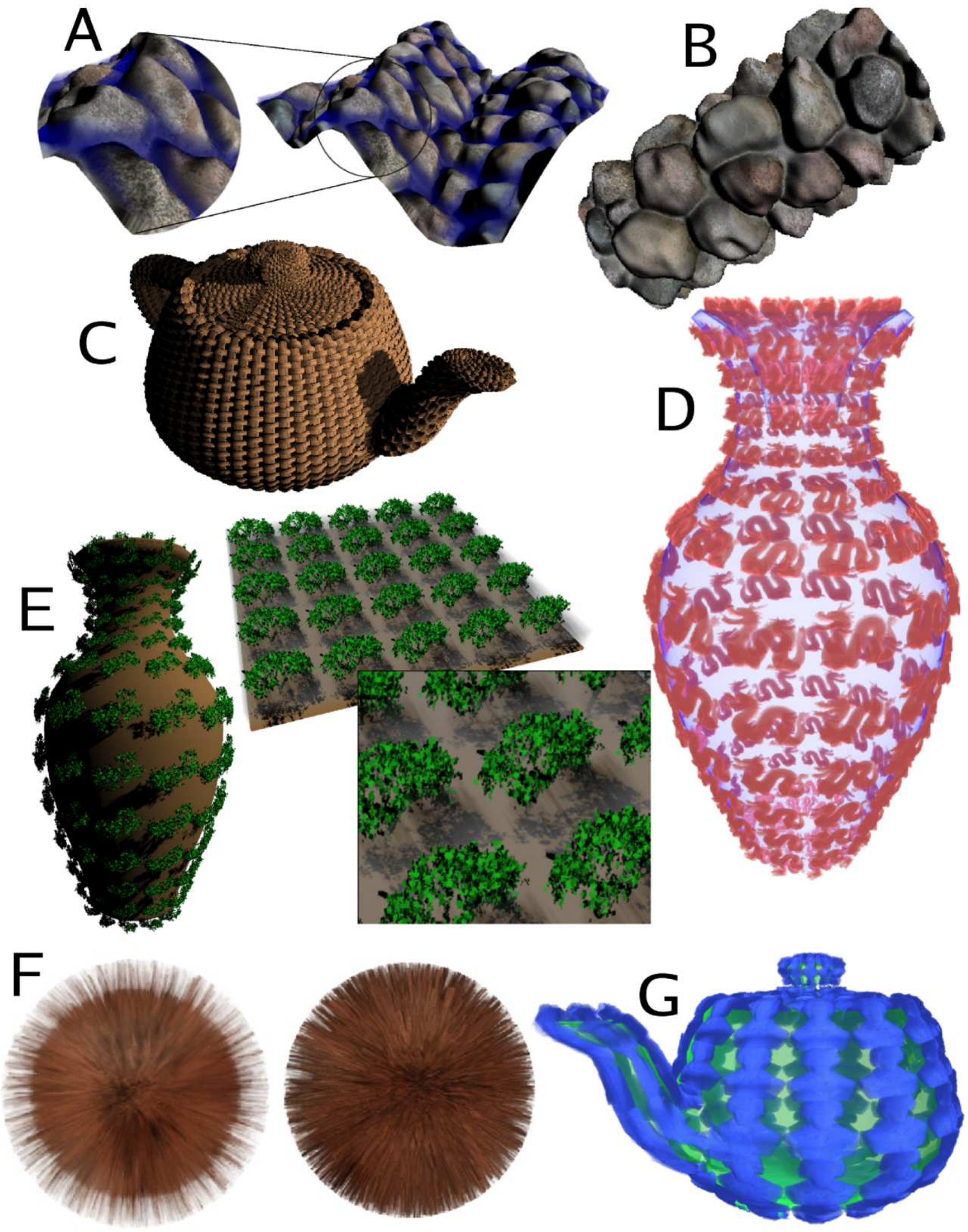
[22] L. Wang, X. Wang, X. Tong, S. Lin, S. Hu, B. Guo, and H.-Y. Shum. View-dependent displacement mapping. In *SIGGRAPH 2003*, pages 334–339, 2003.

[23] X. Wang, X. Tong, S. Lin, S. Hu, B. Guo, and H.-Y. Shum. Generalized displacement maps. In *Rendering Techniques 2004*, 2004.

[24] M. Weiler, M. Kraus, and T. Ertl. Hardware-based view-independent cell projection. In *IEEE Volume Visualization and Graphics*, pages 13–22, 2002.

[25] T. Welsh. Parallax mapping with offset limiting: A per-pixel approximation of uneven surfaces. Infiscape Corp., 2004.

[26] B. Wylie, K. Moreland, L. A. Fisk, and P. Crossno. Tetrahedral projection using vertex shaders. In *IEEE Volume Visualization and Graphics 2002*, pages 7–12, 2002.



Various images rendered at interactive framerates with our technique.