Nonlinear Ray Tracing for Displacement and Shell Mapping

Shinji Ogaki
ZOZO, Inc.
Tokyo, Japan
shinji.ogaki@gmail.com

Figure 1: Our nonlinear ray tracing can render highly detailed geometry without requiring a large amount of memory. The ray tracing performance is 2.46 M rays/s on 20 CPU cores of an Apple M1 Ultra chip and memory usage is 186.0 M bytes.

ABSTRACT

Displacement mapping and shell mapping add fine-scale geometric features to meshes and can significantly enhance the realism of an object's surface representation. Both methods generate geometry within a layer between the base mesh and its offset mesh called a shell. It is not easy to simultaneously achieve high ray tracing performance, low memory consumption, interactive feedback, and ease of implementation, partly because the mapping between shell and texture space is nonlinear. This paper introduces a new efficient approach to perform acceleration structure traversal and intersection tests against microtriangles entirely in texture space by formulating nonlinear rays as degree-2 rational functions. Our method simplifies the implementation of tessellation-free displacement mapping and smooth shell mapping and works even if base mesh triangles are degenerated in *uv* space.

CCS CONCEPTS

• **Computing methodologies** → **Ray tracing**.

KEYWORDS

ray tracing, displacement mapping, shell mapping

SA Conference Papers '23, December 12–15, 2023, Sydney, NSW, Australia

ACM Reference Format:
Shinji Ogaki. 2023. Nonlinear Ray Tracing for Displacement and Shell Mapping. In *SIGGRAPH Asia 2023 Conference Papers (SA Conference Papers '23), December 12–15, 2023, Sydney, NSW, Australia.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3610548.3618199

1 INTRODUCTION

Displacement mapping can add details to a mesh and has been an essential feature for production renderers [Burley et al. 2018; Christensen et al. 2018; Fascione et al. 2018; Georgiev et al. 2018; Kulla et al. 2018]. Shell mapping [Porumbescu et al. 2005] has more expressive power since it can generate intricate mesostructures such as woven threads in a layer between the base mesh and its offset mesh called a shell. The two methods may appear to be different but they can be viewed as essentially the same, since displacement mapping is a method of generating a height field in shell space. Though many studies have been conducted to efficiently realize displacement and shell mapping, there are still open problems. The difficulty originates from the fact that the mapping between shell space and texture space is nonlinear, and thus a rectilinear ray in shell space becomes nonlinear in texture space (Fig.2). The methods for rendering geometry in shell space can be roughly classified into two categories. One is to trace rectilinear rays in world space, and the other is to perform ray marching in texture space.

The most trivial approach to implement displacement mapping is to perform pre-tessellation. Pre-tessellated geometry allows high ray tracing performance, but can consume a significant amount of memory and interactive editing of the displacement content is virtually impossible. Therefore, methods such as out-of-core rendering and on-the-fly tessellation [Benthin et al. 2015; Christensen

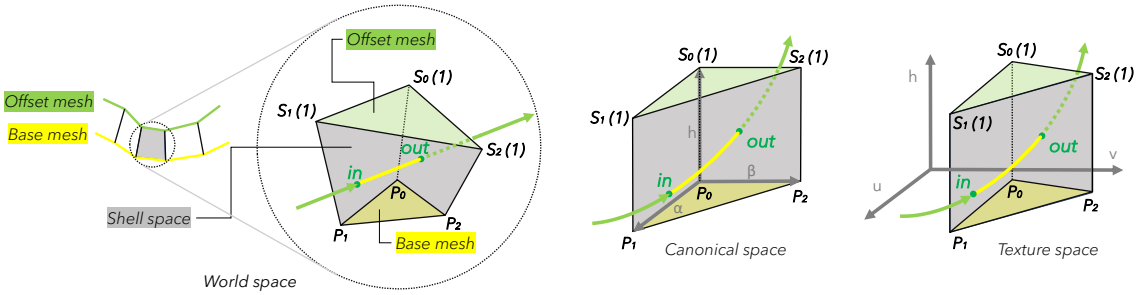

Figure 2: Shell space, canonical space, and texture space. A rectilinear ray in shell space becomes a nonlinear ray in canonical space, and also in texture space.

et al. 2006] have been proposed. Achieving scalability with respect to the number of threads is crucial, but highly optimized implementations of these algorithms are not simple. Recently proposed tessellation-free displacement mapping [Thonat et al. 2021] dramatically reduces memory usage and provides interactive feedback by utilizing a minmax mipmap as an implicit acceleration data structure. Rather than explicitly storing a bounding box at each node of the data structure, conservative bounding boxes are generated on-the-fly using affine arithmetic. While the results are impressive, a few drawbacks exist. First, the number of traversal steps increases due to overlapping bounding boxes generated during traversal. Second, affine approximations have a high degree of design freedom (e.g., when to use the *Min-Range* or *Chebyshev* approximation), requiring in-depth knowledge to balance the tightness of bounding boxes and computational cost. Lastly, ray-microtriangle intersection tests fail if a base mesh triangle degenerates to a point or line in *uv* space, because an inverse matrix is required to calculate barycentric coordinates from texture *uv*s.

Shell mapping, a generalization of displacement mapping, was first introduced by Porumbescu et al. [2005]. It allows generating the instances of arbitrary objects in a shell. Early shell mapping algorithms [Porumbescu et al. 2005; Ritsche 2006] divide prisms that comprise a shell into tetrahedra and perform ray marching by computing the mapping between shell and texture spaces using the barycentric coordinates of a tetrahedron. Tetrahedralization increases the initialization cost, and this piecewise-linear approximation can lead to severe aliasing artifacts. To avoid relying on tetrahedralization, Jeschke et al. [2007] later introduced smooth shell mapping, which performs ray marching by solving a cubic equation at every step. Ritsche [2006] and Jeschke et al. [2007] used a distance map to reduce the number of ray marching steps and achieve real-time rendering. However, rendering photorealistic images with full global illumination is still prohibitive because the rendering performance and the complexity of geometry are bounded by the resolution of a three-dimensional distance map.

To overcome the shortcomings of the previous methods, we introduce a new algorithm that directly traces nonlinear rays in texture space. By formulating a nonlinear ray as a degree-2 rational function, both nonlinear ray-microtriangle and nonlinear ray-axis aligned bounding box (AABB) intersection tests can be solved analytically. Since we aim to render complex scenes on a limited memory budget, we use a minmax mipmap as the acceleration

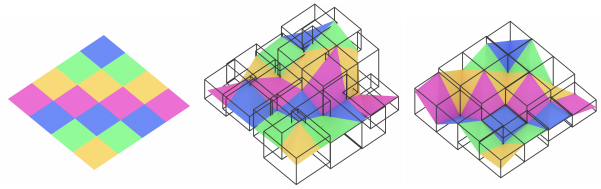

Figure 3: Scalar displacement-mapped mesh and its leaf node AABBs. The AABBs in shell space do overlap (middle). The AABBs in texture space do not overlap at all (right).

structure for displacement mapping, following Thonat et al. [2021]. For shell mapping, we use an implicit quadtree to facilitate repeated placement of the same object along a mesh surface.

Nonlinear ray tracing eliminates the need for affine arithmetic for AABB intersections and expensive and error-sensitive matrix inversions for texture to world-space mappings used in tessellation-free displacement mapping. It also eliminates tetrahedralization and ray marching from shell mapping. Hence our method is intuitive, works even when base mesh triangles are degenerated in *uv* space, and reduces the initialization costs.

2 RELATED WORK

Displacement mapping. Production renderers often apply subdivision to the base mesh and move the vertices of the subdivided mesh by the amount specified by a displacement map [Burley et al. 2018; Christensen et al. 2018; Fascione et al. 2018; Georgiev et al. 2018; Kulla et al. 2018]. If generated microtriangles fit into the main memory, high ray tracing performance can be achieved through pre-tessellation because a high-quality BVH can be built [Fascione et al. 2018]. Otherwise, one must resort to out-of-core rendering or compression schemes.

Lier et al. [2018] used compressed and quantized BVHs to render subdivision surfaces with displacement. Lossy compressed grid primitives [Benthin et al. 2021] represent the input triangle mesh as a set of vertex grids, which implicitly expresses the connectivity of triangle vertices. Each vertex grid is further divided into subgrids with 5×5 vertices and quantized displacement vectors, over which a BVH is built. Micro-meshes [NVIDIA Corporation 2023] are similar in principle, and each base triangle stores compressed scalar values on a barycentric grid. Hardware that natively supports the

rendering of micro-meshes and a dedicated generation algorithm [Maggiordomo et al. 2023] are available. These methods can render unprecedentedly detailed geometry, but it is unclear how they can be extended to shell mapping.

To handle extremely complex geometry without out-of-core rendering, Thonat et al. [2021] proposed a tessellation-free displacement mapping. Their method uses an implicit BVH represented by a minmax mipmap, saving a significant amount of memory. The bounding box of each node is computed by affine arithmetic during traversal. Though this technique can render highly detailed geometry without long preprocessing time, its use of an intersection shader prevents it from taking full advantage of GPUs that support hardware ray tracing, resulting in a wide performance gap relative to pre-tessellation methods. The performance gap is also due to the fact that the generated bounding boxes are conservative and overlap in world space (Fig.3). Furthermore, intersection tests are done in world space, which requires matrix inversions to map texture coordinates to world coordinates (Eq.7-9 in [Thonat et al. 2021]). Thus their method fails when a base mesh triangle is degenerated in uv space.

An alternative way to render complex geometry is on-the-fly tessellation with (multi-resolution) caching [Benthin et al. 2015; Christensen et al. 2006], but it can have scalability issues. The recent rise of deep learning has made it possible to realize displacement mapping using neural nets [Kuznetsov et al. 2022]. However, generated results can be partially blurred, and supporting shell mapping would result in higher training costs, including data preparation, and higher storage costs for networks.

Shell mapping. The original shell mapping algorithm [Porumbescu et al. 2005] divides each prism that constitutes a shell into tetrahedra, and executes ray marching by finding the correspondence between coordinates in shell space and texture space using the barycentric coordinates of tetrahedra, and binary search is used to perform ray-primitive intersection tests. Ritsche [2006] used a three-dimensional distance map to skip empty space in the shell volume and enabled real-time rendering, however, supporting global lighting effects was left for future work. Tetrahedralization not only adds extra initialization costs, but also creates unpleasant artifacts due to piecewise-linear approximations. To reduce such artifacts, Ye et al. [2007] introduced an algorithm to construct a less distorted shell. Jeschke et al. [2007] computed the mapping between shell space and texture space by solving a cubic equation at each ray marching step, removing the dependence on tetrahedralization and ensuring the smooth rendering of objects in shell space. This method also determines the step size by a distance map, and the marching direction is corrected at each step so as not to exceed a specified error.

Nanite. Nanite [Karis et al. 2021] builds a hierarchical level of detail (LOD) structure of triangle clusters, and the resulting structure is represented as a directed acyclic graph (DAG). At rendering time, only the necessary parts of a scene are rasterized by performing a cut on the DAG based on the viewpoint. Benthin and Peters [2023] used a *Nanite*-style LOD selection to ray trace extremely detailed geometry in real time. Both methods work not only for displacement-mapped meshes but also for any high-resolution meshes. However, initialization is costly because it involves splitting and merging clusters and mesh simplification.

$\hat{\boldsymbol{v}}$	normalized vector of vector $\boldsymbol{v}$
$\langle \boldsymbol{a}, \boldsymbol{b} \rangle$	inner product of vectors $\boldsymbol{a}$ and $\boldsymbol{b}$
$\boldsymbol{a} \times \boldsymbol{b}$	cross product of vectors $\boldsymbol{a}$ and $\boldsymbol{b}$
$\boldsymbol{\omega}$	ray direction
t	signed distance along ray direction
h	height parameter
$(1-\alpha-\beta, \alpha, \beta)$	barycentric coordinates
(α, β, h)	canonical space coordinates
(u, v, h)	texture space coordinates
$\boldsymbol{I}(t)$	ray in world space
$\boldsymbol{I}_{\alpha\beta h}(h)$	ray in canonical space
$\boldsymbol{I}_{uvh}(h)$	ray in texture space
$\boldsymbol{P}_i$	i-th vertex of base triangle
$\boldsymbol{N}_i$	normal vector associated with $\boldsymbol{P}_i$
$\boldsymbol{T}_i = (U_i, V_i)$	texture uv coordinates associated with $\boldsymbol{P}_i$
$\boldsymbol{p_i}$	i-th vertex of microtriangle
$\boldsymbol{n}$	normal vector of microtriangle in world space
$\boldsymbol{n}_{\alpha\beta h} = (n_\alpha, n_\beta, n_h)^T$	normal vector of microtriangle in canonical space
$\boldsymbol{n}_{uvh} = (n_u, n_v, n_h)^T$	normal vector of microtriangle in texture space

Table 1: Terms used in the paper.

Nonlinear rays. Nonlinear rays have been successfully used to compute light and sound propagation in nonlinear media. A ray is often divided into piecewise-linear segments, and the direction is corrected each time the ray travels a small step [Cao et al. 2010; Gribble 2021; Suffern and Getto 1991]. Na and Jung [2008] applied curved ray tracing to displacement mapping, but they also used a piecewise-linear approximation. Mo et al. [2016] used circular and parabolic rays to analytically perform acceleration data structure traversal and primitive intersection tests. Nonlinear rays are also useful for artistic purpose [Kerr et al. 2010].

3 BACKGROUND

For subsequent discussions, we first define shell space and texture space, and explain why we need to solve a cubic equation to establish a mapping between shell and texture spaces. Table.1 summarizes the terminology used in the paper.

3.1 Shell Space

Shell space is defined as a layer bounded by the base mesh and its offset mesh (Fig.2). By letting

$$\boldsymbol{S_i}(h) = \boldsymbol{P_i} + h\boldsymbol{N_i},$$

a point in shell space is given as

$$\boldsymbol{S}(\alpha, \beta, h) = (1-\alpha-\beta)\boldsymbol{S_0}(h) + \alpha\boldsymbol{S_1}(h) + \beta\boldsymbol{S_2}(h), \tag{1}$$

where $(1-\alpha-\beta, \alpha, \beta)$ is barycentric coordinates, h is a height parameter, $\triangle \boldsymbol{P_0}\boldsymbol{P_1}\boldsymbol{P_2}$ is a triangle of the base mesh, and $\boldsymbol{N_i}$ is the vertex normal associated with the i-th vertex $\boldsymbol{P_i}$. Within this space, scalar displacement mapping creates a height field, and shell mapping places the instances of arbitrary objects. Hereinafter (α, β, h) is referred to as canonical space coordinates ($0 \leq \alpha, \beta, h \leq 1$ and $\alpha + \beta \leq 1$).

3.2 Texture Space

Texture space coordinates (u, v, h) are obtained by replacing α and β of canonical space coordinates by a texture uv as

$$u = (1 - \alpha - \beta)U_0 + \alpha U_1 + \beta U_2 \tag{2}$$
$$v = (1 - \alpha - \beta)V_0 + \alpha V_1 + \beta V_2, \tag{3}$$

where (U_i, V_i) represents the texture uv coordinates associated with the base mesh triangle vertex $\boldsymbol{P_i}$.

3.3 Mapping between Two Spaces

Converting world coordinates to its corresponding canonical coordinates requires solving a cubic equation [Jeschke et al. 2007]. Let $\boldsymbol{I}$ be a point in world space. Its height parameter h is obtained by solving

$$0 = \langle \boldsymbol{I} - \boldsymbol{S_0}(h), \boldsymbol{N}(h) \rangle, \tag{4}$$

where $\boldsymbol{N}(h)$ is the normal vector of the triangle $\triangle \boldsymbol{S_0}(h)\boldsymbol{S_1}(h)\boldsymbol{S_2}(h)$ given as

$$\boldsymbol{N}(h) = (\boldsymbol{S_1}(h) - \boldsymbol{S_0}(h)) \times (\boldsymbol{S_2}(h) - \boldsymbol{S_0}(h)).$$

Once h is obtained, we can compute α and β by Eq.1. The corresponding texture coordinates are obtained with Eq.2 and 3.

4 NONLINEAR RAY TRACING

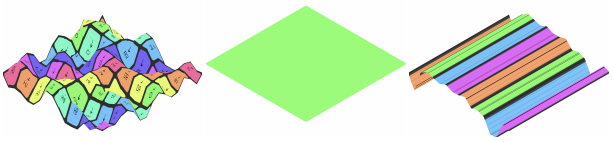

Figure 4: Base mesh degenerated in uv space. The state of the art method [Thonat et al. 2021] cannot render displaced geometry when the base mesh is degenerated to a point (middle) or line (right) in uv space. Our method can render such cases.

To avoid the use of affine arithmetic and ray marching, we use nonlinear ray tracing. In our approach, intersection tests against primitives and BVH traversal are entirely done in texture space.

In this paper, we limit ourselves to triangle primitives. The intersection test between a nonlinear ray and microtriangle requires solving a cubic equation. Though this may seem inefficient, the intention is to avoid matrix inversions and to find intersections even when base mesh triangles degenerate to points or lines in uv space (Fig.4). For displacement mapping, we only need to intersect with two microtriangles per texel because we perform intersection tests in texture space. On the other hand, Thonat et al. [2021] performed intersection tests in world space, and to guarantee watertightness, they clipped microtriangles in each texel and generated up to five microtriangles.

To perform an intersection test against an AABB in texture space requires solving four quadratic equations. This is more expensive than the standard rectilinear ray-AABB intersection test but avoids complex affine approximations to obtain conservative AABBs. We can also handle more complex geometries than the combination of ray marching and a distance map. Additionally, the AABBs of the bottom level acceleration structure (see Sec.4.4) do not overlap in texture space (Fig.3).

4.1 Nonlinear Ray in Canonical Space

Instead of performing ray marching, we directly trace a nonlinear ray by representing it as a rational function whose numerator and denominator have degree two. To obtain the canonical space coordinates (α, β, h) of a point on a ray $\boldsymbol{I}(t) = \boldsymbol{o} + \hat{\boldsymbol{\omega}}t$, we solve

$$\boldsymbol{o} + \hat{\boldsymbol{\omega}}t = (1 - \alpha - \beta)\boldsymbol{S_0}(h) + \alpha \boldsymbol{S_1}(h) + \beta \boldsymbol{S_2}(h). \tag{5}$$

By taking dot products on the both sides with two vectors $\boldsymbol{e_0}$ and $\boldsymbol{e_1}$ perpendicular to $\hat{\boldsymbol{\omega}}$, we remove $\hat{\boldsymbol{\omega}}t$ from the above equation:

$$\begin{cases} \langle \boldsymbol{e_0}, \boldsymbol{E_0}(h) \rangle \alpha + \langle \boldsymbol{e_0}, \boldsymbol{E_1}(h) \rangle \beta = \langle \boldsymbol{e_0}, \boldsymbol{o} - \boldsymbol{S_0}(h) \rangle \\ \langle \boldsymbol{e_1}, \boldsymbol{E_0}(h) \rangle \alpha + \langle \boldsymbol{e_1}, \boldsymbol{E_1}(h) \rangle \beta = \langle \boldsymbol{e_1}, \boldsymbol{o} - \boldsymbol{S_0}(h) \rangle \end{cases} \tag{6}$$

where $\boldsymbol{E_i}(h) = \boldsymbol{S_{i+1}}(h) - \boldsymbol{S_0}(h)$, and $\boldsymbol{e_0} \perp \boldsymbol{e_1}$

Parametric curve of height. If we solve Eq.6 for α and β, a ray in canonical space can be represented as a parametric curve of h. It is easy to show that both α and β are quadratic rational functions of h. Thus we can write a ray in the form of

$$\boldsymbol{I}_{\alpha\beta h}(h) = \left(\frac{\alpha_2 h^2 + \alpha_1 h + \alpha_0}{d_2 h^2 + d_1 h + d_0}, \frac{\beta_2 h^2 + \beta_1 h + \beta_0}{d_2 h^2 + d_1 h + d_0}, h \right), \tag{7}$$

where $\alpha_i, \beta_i, d_i \in \mathbb{R}$.

Parametric curve via rational conic. Solving one of Eq.6 for h and substituting it into the other, the relation between α and β can be represented as a conic section. It is known that a conic section can be converted to a quadratic rational Bézier curve [Cox et al. 2007] so α and β can be expressed as quadratic rational functions. Unfortunately, this representation makes the height parameter h another quadratic rational function, resulting in more complicated calculations. We thus parameterize a nonlinear ray by h with Eq.7.

4.2 Nonlinear Ray in Texture Space

To perform nonlinear ray tracing in texture space, a ray must be represented in texture space. With Eq.7, the uv coordinates of a point on a ray can be written as

$$\boldsymbol{T_0} + (\boldsymbol{T_1} - \boldsymbol{T_0}) \frac{\alpha_2 h^2 + \alpha_1 h + \alpha_0}{d_2 h^2 + d_1 h + d_0} + (\boldsymbol{T_2} - \boldsymbol{T_0}) \frac{\beta_2 h^2 + \beta_1 h + \beta_0}{d_2 h^2 + d_1 h + d_0},$$

where $\boldsymbol{T_i} = (U_i, V_i)^T$. By letting

$$u_i = (d_i - \alpha_i - \beta_i)U_0 + \alpha_i U_1 + \beta_i U_2$$
$$v_i = (d_i - \alpha_i - \beta_i)V_0 + \alpha_i V_1 + \beta_i V_2,$$

we can represent a ray in texture space as

$$\boldsymbol{I}_{uvh}(h) = \left(\frac{u_2 h^2 + u_1 h + u_0}{d_2 h^2 + d_1 h + d_0}, \frac{v_2 h^2 + v_1 h + v_0}{d_2 h^2 + d_1 h + d_0}, h \right). \tag{8}$$

Note that the denominators remain unchanged from Eq.7.

4.3 Intersection Tests in Texture Space

We directly perform nonlinear ray-primitive intersection tests in texture space. Finding the intersections of a nonlinear ray and plane is a fundamental part of our method because it is used for intersection tests against microtriangles and AABBs.

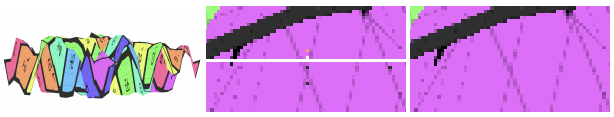

Figure 5: Eq.6 alone fails to render properly where the denominator is close to 0. White pixels are where the intersection tests failed (middle). Using Eq.11 prevents this error (right).

Nonlinear ray-microtriangle intersection. The plane containing a microtriangle $\triangle \boldsymbol{p}_0\boldsymbol{p}_1\boldsymbol{p}_2$ is given as

$$0 = n_u u + n_v v + n_h h + K_{uvh}, \tag{9}$$

where $\boldsymbol{n}_{uvh} = (n_u, n_v, n_h)^T$ is the normal vector of the microtriangle and K_{uvh} is a constant. Our method first seeks the values of h. Substituting Eq.8 into Eq.9 and clearing the denominator yields a cubic equation of h. After solving this, we find α and β. Unfortunately, we cannot compute α and β robustly with Eq.6 alone because the determinant can be 0 or close to 0 (Fig.5). For example, it becomes 0 if $h_{in} = h_{out}$, where h_{in} and h_{out} are the values of h at the point where a ray first hits the prism and the point where it last leaves the prism, respectively. (Note that we can find h even if $h_{in} = h_{out}$ by clearing the denominator.) To yield another linear equation of α and β, we convert the plane from texture space to canonical space by substituting Eq.2 and 3 into Eq.9:

$$0 = n_\alpha \alpha + n_\beta \beta + n_h h + K_{\alpha\beta h} \tag{10}$$

where

$$\begin{aligned} n_\alpha &= n_u(U_1 - U_0) + n_v(V_1 - V_0) \\ n_\beta &= n_u(U_2 - U_0) + n_v(V_2 - V_0) \\ K_{\alpha\beta h} &= U_0 n_u + V_0 n_v + K_{uvh} \end{aligned}$$

Combined with Eq.6, we now have three equations of α and β:

$$\begin{cases} \langle \boldsymbol{e}_0, \boldsymbol{E}_0(h)\rangle\alpha + \langle \boldsymbol{e}_0, \boldsymbol{E}_1(h)\rangle\beta = \langle \boldsymbol{e}_0, \boldsymbol{o} - \boldsymbol{S}_0(h)\rangle \\ \langle \boldsymbol{e}_1, \boldsymbol{E}_0(h)\rangle\alpha + \langle \boldsymbol{e}_1, \boldsymbol{E}_1(h)\rangle\beta = \langle \boldsymbol{e}_1, \boldsymbol{o} - \boldsymbol{S}_0(h)\rangle \\ n_\alpha\alpha + n_\beta\beta = -n_h h - K_{\alpha\beta h} \end{cases} \tag{11}$$

We find the intersection by choosing two equations from Eq.11 so that the absolute value of the determinant becomes the largest. Note that dividing the both sides of Eq.9 by $|\boldsymbol{n}_{uvh}|$ beforehand improves robustness because microtriangles are generally very small and so is $\boldsymbol{n}_{uvh}$. Obtaining u and v from α and β is trivial. Lastly, we check if the intersection is inside both the microtriangle and the prism.

Signed distance. The signed distance parameter in world space t can be obtained by taking dot products on the both sides of Eq.5 with $\hat{\boldsymbol{\omega}}$:

$$t = \langle \hat{\boldsymbol{\omega}}, (1 - \alpha - \beta)\boldsymbol{S}_0(h) + \alpha\boldsymbol{S}_1(h) + \beta\boldsymbol{S}_2(h) - \boldsymbol{o}\rangle$$

Normal vector. The normal vector $\boldsymbol{n}_{uvh}$ cannot be used for the actual shading because it is defined in texture space. The normal vector in world space $\boldsymbol{n}$ can be obtained by applying a linear transformation to $\boldsymbol{n}_{\alpha\beta h} = \left(n_\alpha, n_\beta, n_h\right)^T$:

$$\boldsymbol{n} = \boldsymbol{M}^{-T}\boldsymbol{n}_{\alpha\beta h} \tag{12}$$

where

$$\boldsymbol{M} = \begin{pmatrix} | & | & | \\ \boldsymbol{E}_0(h) & \boldsymbol{E}_1(h) & \boldsymbol{N}_s \\ | & | & | \end{pmatrix}$$

and

$$\boldsymbol{N}_s = \boldsymbol{N}_0(1 - \alpha - \beta) + \alpha\boldsymbol{N}_1 + \beta\boldsymbol{N}_2$$

In practice, $\boldsymbol{n}$ is normalized before shading. We thus use the adjugate matrix instead of the transpose of the inverse matrix and avoid the division by the determinant:

$$\boldsymbol{n} = \mathrm{adj}(\boldsymbol{M})\boldsymbol{n}_{\alpha\beta h} \tag{13}$$

Note that $\boldsymbol{n}$ can be obtained even if the prism degenerates to a plane (e.g., $\boldsymbol{E}_0(h) = \boldsymbol{E}_1(h) \neq \boldsymbol{N}_s$).

We now examine when $\boldsymbol{n}$ cannot be determined by Eq.13. The first case is $0 = \mathrm{adj}(\boldsymbol{M})$. In this case a ray does not intersect the microtriangle because the prism is degenerated to a line or point in world space. The second case is when $0 = n_\alpha = n_\beta = n_h$. Since $0 < |\boldsymbol{n}_{uvh}|$, this case only happens when the base triangle is degenerated in uv space and $0 = n_h$. In such a situation, the microtriangle and a ray are parallel, and they do not intersect.

When $0 \neq n_h$, which is always true for scalar displacement mapping, we can take a slightly less expensive approach. In general the normal vector of a parametric surface $\boldsymbol{f}(\alpha, \beta)$ can be computed as $\partial\boldsymbol{f}/\partial\alpha \times \partial\boldsymbol{f}/\partial\beta$. To apply this, we solve Eq.10 for h and substitute it into Eq.1. The partial derivatives are then given as

$$\begin{aligned} \frac{\partial \boldsymbol{S}(\alpha,\beta)}{\partial\alpha} &= \boldsymbol{E}_0(h) - \frac{n_\alpha}{n_h}\boldsymbol{N}_s \\ \frac{\partial \boldsymbol{S}(\alpha,\beta)}{\partial\beta} &= \boldsymbol{E}_1(h) - \frac{n_\beta}{n_h}\boldsymbol{N}_s. \end{aligned}$$

4.4 Nonlinear Ray Traversal

We use a multi-level acceleration structure. The top level acceleration structure (TLAS) is a BVH built for prisms spanning the base and offset meshes. Each leaf of the TLAS stores an implicit acceleration data structure, referred to as a bottom level acceleration structure (BLAS). For displacement mapping, the BLAS is a minmax mipmap and each leaf of which stores two microtriangles (Fig.6, middle). For shell mapping, the BLAS is an implicit quadtree that consumes no memory, and all the leaves of the quadtree refer to the BVH built for an object to be instanced in a shell (Fig.6, right). Shell mapping does not necessarily require a quadtree, but we use it to facilitate the generation of textiles and knitted fabrics by tiling the same object. Quadtree traversal is executed in the same manner as minmax mipmap traversal. When a ray intersects a leaf of the quadtree, we shift it in uv space, and then start traversing the BVH built for the instanced object. For the general concepts and terms of the BVH, we refer the readers to the survey by Meister et al. [2021].

Traversing TLAS. We traverse the TLAS with rectilinear rays so the standard traversal routine can be used. If a rectilinear ray hits a prism, we start traversing the associated BLAS. Otherwise, we continue traversing the TLAS. There is no difference in TLAS traversal between displacement mapping and shell mapping. The sides of each prism are bilinear patches, and we use the method by Reshetov [2019] for ray-bilinear patch intersection tests.

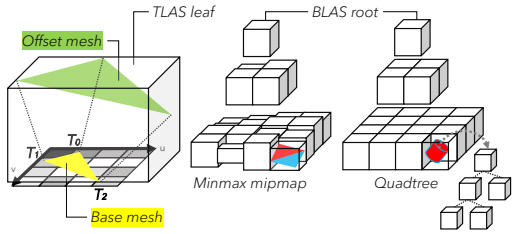

Figure 6: TLAS and BLAS. Each texel of the finest level minmax mipmap contains two microtriangles, and each leaf node of the quadtree stores a reference to the BVH built for an object instanced in the shell.

Traversing BLAS. BLAS traversal is done with nonlinear rays in texture space. At each node we perform a nonlinear ray-AABB intersection test. We simply intersect a ray against the six faces of an AABB. The intersections of the ray and planes $u = \{u_{min}, u_{max}\}$ are obtained by solving quadratic equations

$$\{u_{min}, u_{max}\} = \frac{u_2 h^2 + u_1 h + u_0}{d_2 h^2 + d_1 h + d_0}.$$

The intersections of the ray and planes $v = \{v_{min}, v_{max}\}$ are obtained in the same manner. For the planes $h = \{h_{min}, h_{max}\}$, it is unnecessary to solve quadratic equations because the intersections are readily available from Eq.8. When the denominator of Eq.8 is 0, the ray and planes $h = \{h_{min}, h_{max}\}$ are parallel and do not intersect. If any of the intersections are inside the AABB, we descend into child nodes.

5 IMPLEMENTATION

This section describes the implementation details. Our goals are to reduce the time to first pixel so that users can have interactive feedback, and to make rendering as fast as possible.

5.1 Initialization

The tasks needed during the initialization phase are:

- for tessellation-free displacement mapping
  - minmax mipmap generation
- for shell mapping
  - BVH construction for instanced object
- for both
  - BVH construction over prisms
  - root selection (optional, see Sec.5.4)

We do not explicitly generate prisms that span the base and offset meshes. They are constructed on-the-fly. The memory consumption and initialization cost of our displacement mapping implementation are identical to those of the method by Thonat et al. [2021]. Our shell mapping implementation requires the construction of a BVH in texture space for an object to be instanced but does not require tetrahedralization and the creation of a distance map.

5.2 Acceleration Data Structure

The TLAS is a 4-ary BVH built with the greedy top-down plane-sweeping algorithm [MacDonald and Booth 1990] using the surface area heuristic, and a single prism is stored at each leaf node simply assuming that each prism has the same cost. The bounding boxes are axis-aligned and no spatial splitting is used. The BLAS is an implicit 4-ary BVH or quadtree. The BVH for an instanced object is built in the same manner as the TLAS. We do not use SIMD ray traversal for nonlinear rays, so there is a room for performance improvement.

5.3 Polynomial Solver

Our ray-microtriangle intersection requires solving a cubic equation. Reshetov [2022] implements a direct solver for ray-ribbon intersections tests, however, we adopt the solver by Yuksel [2022] because of its simplicity. We use a slightly modified version that explicitly calls fused multiply-adds whenever possible.

5.4 Optimization

We use the optimization techniques used in the existing methods and their improvements.

Root selection and texel discard. For displacement mapping, we adopt two optimization techniques by Thonat et al. [2021]. If the input base mesh is reasonably tessellated, traversing from the lowest resolution mipmap is not efficient. We can traverse from the level and texel at which a base triangle is fully enclosed. This accelerates rendering at the expense of precomputation and storage. We also avoid traversing texels that do not overlap a base mesh triangle. These techniques can also be applied to shell mapping.

Traversal order. While Thonat et al. [2021] decided the traversal order based on the direction of a ray in uv space, we sort child nodes based on increasing hit distance.

Range reduction. When executing a ray-prism intersection test in TLAS traversal, we record the distance parameters of the first entry and last exit points t_{in} and t_{out}. If the signed distances to the intersections of a nonlinear ray and the AABB of a BLAS node are outside of the range $[t_{in}, t_{out}]$, we skip entering the node. We also record the height parameters of the entry and exit points h_{in} and h_{out}. These are used to limit the range of h when solving a cubic equation.

LODs. For displacement mapping, we do not implement LODs [Thonat et al. 2021] because simply selecting mipmap level based on the distance is not necessarily an ideal solution. When applying LODs to high frequency-geometries (e.g., Fig.7 (c)), some parts should be treated as transparent. Instead, if the min and max values of a texel of the current mipmap are equal, we immediately terminate traversing the BLAS and perform a rectilinear ray-flat rectangle intersection test. Similarly, if the h coordinates of the three vertices of a microtriangle are the same, we switch to the standard rectilinear ray-flat triangle intersection test. When h is constant, a plane becomes flat not only in texture space but also in world space (see Eq.1). For shell mapping, it is not difficult to support LODs by implementing, for example, traversal shaders [Lee et al. 2019].

		base mesh		displacement map					
		triangles	memory	resolution	memory	tiling	microtriangles	total memory	performance
displacement mapping (pre-tessellation)	(a) Mountain	2	-	4096^2	-	1×1	33, 554, 432	2.6 GB	**24.92** M rays/s
	(b) Cylinder	512	-	4096^2	-	1×1	33, 554, 432	2.6 GB	**23.64** M rays/s
	(c) Torus (Fur)	8, 192	-	256^2	-	8×8	8, 388, 608	0.7 GB	0.49 M rays/s
displacement mapping (ours)	(a) Mountain	2	0.4 kB	4096^2	42.7 MB	1×1	33, 554, 432	42.7 MB	7.74 M rays/s
	(b) Cylinder	512	35.4 kB	4096^2	42.7 MB	1×1	33, 554, 432	42.7 MB	5.00 M rays/s
	(c) Torus (Fur)	8, 192	561.0 kB	256^2	0.2 MB	8×8	8, 388, 608	0.7 MB	**1.09** M rays/s
		base mesh		instanced object					
		triangles	memory	triangles	memory	tiling	microtriangles	total memory	performance
shell mapping	(d) Torus (Feathers)	2, 400	0.3 MB	28, 438	3.0 MB	64×128	232, 964, 096	3.3 MB	1.00 M rays/s
	(e) Scales	14, 352	1.8 MB	18, 895	1.9 MB	16×16	96, 742, 400	3.7 MB	2.11 M rays/s
	(f) Dress	174, 702	24.2 MB	16, 896	1.7 MB	16×16	267, 778, 508	25.9 MB	2.06 M rays/s
	(g) Cornell boxes	722	0.1 MB	2, 894	0.3 MB	16×16	740, 864	0.4 MB	9.60 M rays/s
	(h) Metal bunnies	722	0.1 MB	145, 780	20.3 MB	16×16	37, 319, 680	20.4 MB	3.49 M rays/s

Table 2: Statistics of the test scenes in Fig.7. Memory consumption includes mipmaps, BVH, vertex data (positions, normals, *uv*s), and connectivity information.

6 RESULTS

We implemented the described method in our research renderer. All the images except Fig.8 and Fig.9 are rendered with single precision. All renderings and timings were performed on 20 CPU cores of an Apple M1 Ultra chip.

First, we rendered several scenes using our method. The statistics are available in Table 2. Fig.7 (a-c) show displacement-mapped objects. For (a) and (b) our method achieves from 20 to 30% of the speed of the rendering of the pre-tessellated geometry. For (c), our method uses only a thousandth of the memory and renders more than twice as fast because AABBs in texture space behave like oriented bounding boxes [Vitsas et al. 2023; Woop et al. 2014] in world space. Fig.7 (d-h) show shell-mapped objects. A large number of identical objects can be easily tiled along the surface of a base mesh thanks to a quadtree. The ray tracing performance ranges from 1.00 to 9.60 M rays/s. In Fig.1, we use both displacement mapping and shell mapping to create a very complex scene consisting of 757, 071, 872 microtriangles.

Next, we investigated when the lack of precision can be problematic. In Fig.8, displacement mapping is applied to a torus. With single precision, some rays slip through the geometry. Rendering with double precision does not cause such problems, but there is no guarantee that the use of double precision gives perfect results. In Fig.9, we applied shell mapping to a quadrangle and instanced a Cornell box-like object in its shell. Our method can render this test scene with single and double precision if the base mesh degenerates in *uv* or world space and even if the vertex normals are oriented to cause self-intersections.

Though the results are not compared with the existing methods, we would like to emphasize that the proposed method can render scenes containing base mesh triangles degenerated in *uv* space, which is not possible with the state-of-the-art method by Thonat et al. [2021]. For shell mapping, it is possible to apply various techniques developed for the BVH such as a spatial-temporal BVH [Woop et al. 2017] and programmable instance [Lee et al. 2019] since we use the BVH as an acceleration data structure. This is a major advantage of our method.

7 CONCLUSION

We proposed the use of nonlinear ray tracing for displacement and shell mapping. Though our method overcomes the shortcomings of the existing methods, our approach has some limitations and problems to be addressed.

Precision issue. Our method is not completely watertight due to numerical errors (Fig.8). We would like to guarantee watertightness in the same spirits of the watertight ray-triangle intersection test [Woop et al. 2013] and robust BVH traversal [Ize 2013]. For a bounced ray, shifting its origin [Wächter and Binder 2019; Wald 2021] may further improve results.

Surface area computation. Pre-tessellation consumes considerable amount of memory but it has clear benefits. For example, it makes surface sampling easy because one can access all microtriangles existing in a scene. In our method, obtaining the surface area of a displacement-mapped object is complex because one must compute integrals.

Self-intersection. Our method works when self-intersection occurs (Fig.8), but care should be taken, e.g. to avoid light leaks when one side of a material is emissive.

Vector displacement mapping. Our method does not support vector displacement mapping. However, for example, when generating fur, we can control the flow by varying the orientations of the vertex normals of a base mesh.

Discontinuity and distortion. Our implementation realizes smooth shell mapping [Jeschke et al. 2007] which is C^1 continuous within every prism and C^0 continuous between prisms. We believe that supporting curved shell mapping [Jeschke et al. 2007] addresses the discontinuity issue because the resulting surface maintains tangent continuity at the prism boundaries. Our method creates cracks where *uv*s are discontinuous, which is also a limitation of the previous work by Thonat et al. [2021]. The method by Ye et al. [2007] can reduce the distortion of prisms, and the method by Zirr and Ritschel [2019] can mitigate the distortion of color textures.

ACKNOWLEDGMENTS

The author thank anonymous reviewers for their valuable comments, Brian Budge for proofreading and insightful comments, and Kentaro Suzuki and Yuki Ozawa for their feedback, and Marcos Fajardo for finding typos. The height maps used in Fig.1 and Fig.8 are courtesy of Rob Tuytel, and the one in Fig.7 (b) is of Jan Trubač.

REFERENCES

Carsten Benthin and Christoph Peters. 2023. Real-Time Ray Tracing of Micro-Poly Geometry with Hierarchical Level of Detail. *Computer Graphics Forum* 42, 8 (2023), e14868. https://doi.org/10.1111/cgf.14868 arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1111/cgf.14868

Carsten Benthin, Karthik Vaidyanathan, and Sven Woop. 2021. Ray Tracing Lossy Compressed Grid Primitives. In *Eurographics 2021 - Short Papers*, Holger Theisel and Michael Wimmer (Eds.). The Eurographics Association. https://doi.org/10.2312/egs.20211009

Carsten Benthin, Sven Woop, Matthias Nießner, Kai Selgrad, and Ingo Wald. 2015. Efficient Ray Tracing of Subdivision Surfaces Using Tessellation Caching. In *Proceedings of the 7th Conference on High-Performance Graphics* (Los Angeles, California) *(HPG '15)*. Association for Computing Machinery, New York, NY, USA, 5–12. https://doi.org/10.1145/2790060.2790061

Brent Burley, David Adler, Matt Jen-Yuan Chiang, Hank Driskill, Ralf Habel, Patrick Kelly, Peter Kutz, Yining Karl Li, and Daniel Teece. 2018. The Design and Evolution of Disney's Hyperion Renderer. *ACM Transaction on Graphics* 37, 3 (2018), 33:1–33:22.

Chen Cao, Zhong Ren, Baining Guo, and Kun Zhou. 2010. Interactive Rendering of Non-Constant, Refractive Media Using the Ray Equations of Gradient-Index Optics. In *Proceedings of the 21st Eurographics Conference on Rendering* (Saarbrücken, Germany) *(EGSR'10)*. Eurographics Association, Goslar, DEU, 1375–1382. https://doi.org/10.1111/j.1467-8659.2010.01733.x

Per Christensen, Julian Fong, Jonathan Shade, Wayne Wooten, Brenden Schubert, Andrew Kensler, Stephen Friedman, Charlie Kilpatrick, Cliff Ramshaw, Marc Bannister, Brenton Rayner, Jonathan Brouillat, and Max Liani. 2018. RenderMan: An Advanced Path-Tracing Architecture for Movie Rendering. *ACM Transanctions on Graphics* 37, 3 (2018), 30:1–30:21.

Per H. Christensen, Julian Fong, David M. Laur, and Dana Batali. 2006. Ray Tracing for the Movie 'Cars'. In *2006 IEEE Symposium on Interactive Ray Tracing*. 1–6. https://doi.org/10.1109/RT.2006.280208

David Cox, John Little, and Donal O'Shea. 2007. *Ideals, Varieties, and Algorithms. An Introduction to Computational Algebraic Geometry and Commutative Algebra*. Springer New York, NY. https://link.springer.com/book/10.1007/978-0-387-35651-8

Luca Fascione, Johannes Hanika, Mark Leone, Marc Droske, Jorge Schwarzhaupt, Tomáš Davidovič, Andrea Weidlich, and Johannes Meng. 2018. Manuka: A Batch-Shading Architecture for Spectral Path Tracing in Movie Production. *ACM Transactions on Graphics* 37, 3 (2018), 31:1–31:18.

Iliyan Georgiev, Thiago Ize, Mike Farnsworth, Ramón Montoya-Vozmediano, Alan King, Brecht Van Lommel, Angel Jimenez, Oscar Anson, Shinji Ogaki, Eric Johnston, Adrien Herubel, Declan Russell, Frédéric Servant, and Marcos Fajardo. 2018. Arnold: A Brute-Force Production Path Tracer. *ACM Transactions on Graphics* 37, 3 (2018), 32:1–32:12.

Christiaan Gribble. 2021. *Curved Ray Traversal*. Apress, Berkeley, CA, 569–597. https://doi.org/10.1007/978-1-4842-7185-8_36

Thiago Ize. 2013. Robust BVH Ray Traversal. *Journal of Computer Graphics Techniques (JCGT)* 2, 2 (19 July 2013), 12–27. http://jcgt.org/published/0002/02/02/

Stefan Jeschke, Stephan Mantler, and Michael Wimmer. 2007. Interactive Smooth and Curved Shell Mapping. In *Proceedings of the 18th Eurographics Conference on Rendering Techniques* (Grenoble, France) *(EGSR'07)*. Eurographics Association, Goslar, DEU, 351–360.

Brian Karis, Rune Stubbe, and Graham Wihlidal. 2021. A Deep Dive into Nanite Virtualized Geometry. In *ACM SIGGRAPH*.

William B. Kerr, Fabio Pellacini, and Jonathan D. Denning. 2010. Bendylights: Artistic Control of Direct Illumination by Curving Light Rays. In *Proceedings of the 21st Eurographics Conference on Rendering* (Saarbrücken, Germany) *(EGSR'10)*. Eurographics Association, Goslar, DEU, 1451–1459. https://doi.org/10.1111/j.1467-8659.2010.01742.x

Christopher Kulla, Alejandro Conty, Clifford Stein, and Larry Gritz. 2018. Sony Pictures Imageworks Arnold. *ACM Transactions on Graphics* 37, 3 (2018), 29:1–29:18.

Alexandr Kuznetsov, Xuezheng Wang, Krishna Mullia, Fujun Luan, Zexiang Xu, Milos Hasan, and Ravi Ramamoorthi. 2022. Rendering Neural Materials on Curved Surfaces. In *ACM SIGGRAPH 2022 Conference Proceedings* (Vancouver, BC, Canada) *(SIGGRAPH '22)*. Association for Computing Machinery, New York, NY, USA, Article 9, 9 pages. https://doi.org/10.1145/3528233.3530721

Won-Jong Lee, Gabor Liktor, and Karthik Vaidyanathan. 2019. Flexible Ray Traversal with an Extended Programming Model. In *SIGGRAPH Asia 2019 Technical Briefs* (Brisbane, QLD, Australia) *(SA '19)*. Association for Computing Machinery, New York, NY, USA, 17–20. https://doi.org/10.1145/3355088.3365149

Alexander Lier, Magdalena Martinek, Marc Stamminger, and Kai Selgrad. 2018. A High-Resolution Compression Scheme for Ray Tracing Subdivision Surfaces with Displacement. *Proc. ACM Comput. Graph. Interact. Tech.* 1, 2, Article 33 (aug 2018), 17 pages. https://doi.org/10.1145/3233308

David J. MacDonald and Kellogg S. Booth. 1990. Heuristics for Ray Tracing Using Space Subdivision. *Vis. Comput.* 6, 3 (may 1990), 153–166. https://doi.org/10.1007/BF01911006

Andrea Maggiordomo, Henry Moreton, and Marco Tarini. 2023. Micro-Mesh Construction. *ACM Trans. Graph.* 42, 4, Article 121 (jul 2023), 18 pages. https://doi.org/10.1145/3592440

Daniel Meister, Shinji Ogaki, Carsten Benthin, Michael J. Doyle, Michael Guthe, and Jiří Bittner. 2021. A Survey on Bounding Volume Hierarchies for Ray Tracing. *Computer Graphics Forum* 40, 2 (2021), 683–712. https://doi.org/10.1111/cgf.142662 arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1111/cgf.142662

Qi Mo, Hengchin Yeh, and Dinesh Manocha. 2016. Tracing Analytic Ray Curves for Light and Sound Propagation in Non-Linear Media. *IEEE Transactions on Visualization and Computer Graphics* 22, 11 (2016), 2493–2506. https://doi.org/10.1109/TVCG.2015.2509996

Kyung-Gun Na and Moon-Ryul Jung. 2008. Curved Ray-Casting for Displacement Mapping in the GPU. In *Advances in Multimedia Modeling*, Shin'ichi Satoh, Frank Nack, and Minoru Etoh (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 348–357.

NVIDIA Corporation. 2023. Micro-Mesh - Basics. Online Accessed: May 5th 2023. https://developer.download.nvidia.com/ProGraphics/nvpro-samples/slides/Micro-Mesh_Basics.pdf

Serban D. Porumbescu, Brian Budge, Louis Feng, and Kenneth I. Joy. 2005. Shell Maps. In *ACM SIGGRAPH 2005 Papers* (Los Angeles, California) *(SIGGRAPH '05)*. Association for Computing Machinery, New York, NY, USA, 626–633. https://doi.org/10.1145/1186822.1073239

Alexander Reshetov. 2019. *Cool Patches: A Geometric Approach to Ray/Bilinear Patch Intersections*. Apress, Berkeley, CA, 95–109. https://doi.org/10.1007/978-1-4842-4427-2_8

Alexander Reshetov. 2022. Ray/Ribbon Intersections. *Proc. ACM Comput. Graph. Interact. Tech.* 5, 3, Article 28 (jul 2022), 22 pages. https://doi.org/10.1145/3543862

Nico Ritsche. 2006. Real-Time Shell Space Rendering of Volumetric Geometry. In *Proceedings of the 4th International Conference on Computer Graphics and Interactive Techniques in Australasia and Southeast Asia* (Kuala Lumpur, Malaysia) *(GRAPHITE '06)*. Association for Computing Machinery, New York, NY, USA, 265–274. https://doi.org/10.1145/1174429.1174477

Kevin G. Suffern and Phillip H. Getto. 1991. *Ray Tracing Gradient Index Lenses*. Springer-Verlag, Berlin, Heidelberg, 317–331.

Theo Thonat, Francois Beaune, Xin Sun, Nathan Carr, and Tamy Boubekeur. 2021. Tessellation-Free Displacement Mapping for Ray Tracing. *ACM Trans. Graph.* 40, 6, Article 282 (dec 2021), 16 pages. https://doi.org/10.1145/3478513.3480535

N. Vitsas, I. Evangelou, G. Papaioannou, and A. Gkaravelis. 2023. Parallel Transformation of Bounding Volume Hierarchies into Oriented Bounding Box Trees. *Computer Graphics Forum* 42, 2 (2023), 245–254. https://doi.org/10.1111/cgf.14758 arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1111/cgf.14758

Carsten Wächter and Nikolaus Binder. 2019. *A Fast and Robust Method for Avoiding Self-Intersection*. Apress, Berkeley, CA, 77–85. https://doi.org/10.1007/978-1-4842-4427-2_6

Ingo Wald. 2021. *Improving Numerical Precision in Intersection Programs*. Apress, Berkeley, CA, 545–550. https://doi.org/10.1007/978-1-4842-7185-8_34

Sven Woop, Attila T. Áfra, and Carsten Benthin. 2017. STBVH: A Spatial-Temporal BVH for Efficient Multi-Segment Motion Blur. In *Proceedings of High Performance Graphics* (Los Angeles, California) *(HPG '17)*. Association for Computing Machinery, New York, NY, USA, Article 8, 8 pages. https://doi.org/10.1145/3105762.3105779

Sven Woop, Carsten Benthin, and Ingo Wald. 2013. Watertight Ray/Triangle Intersection. *Journal of Computer Graphics Techniques (JCGT)* 2, 1 (28 June 2013), 65–82. http://jcgt.org/published/0002/01/05/

Sven Woop, Carsten Benthin, Ingo Wald, Gregory S. Johnson, and Eric Tabellion. 2014. Exploiting Local Orientation Similarity for Efficient Ray Traversal of Hair and Fur. In *Eurographics/ ACM SIGGRAPH Symposium on High Performance Graphics*, Ingo Wald and Jonathan Ragan-Kelley (Eds.). The Eurographics Association. https://doi.org/10.2312/hpg.20141092

Kai Ye, Kun Zhou, Zhigeng Pan, Yiying Tong, and Baining Guo. 2007. Low Distortion Shell Map Generation. In *2007 IEEE Virtual Reality Conference*. 203–208. https://doi.org/10.1109/VR.2007.352482

Cem Yuksel. 2022. High-Performance Polynomial Root Finding for Graphics. *Proc. ACM Comput. Graph. Interact. Tech.* 5, 3, Article 27 (jul 2022), 15 pages. https://doi.org/10.1145/3543865

Tobias Zirr and Tobias Ritschel. 2019. Distortion-Free Displacement Mapping. *Computer Graphics Forum* 38, 8 (2019), 53–62. https://doi.org/10.1111/cgf.13760 arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1111/cgf.13760

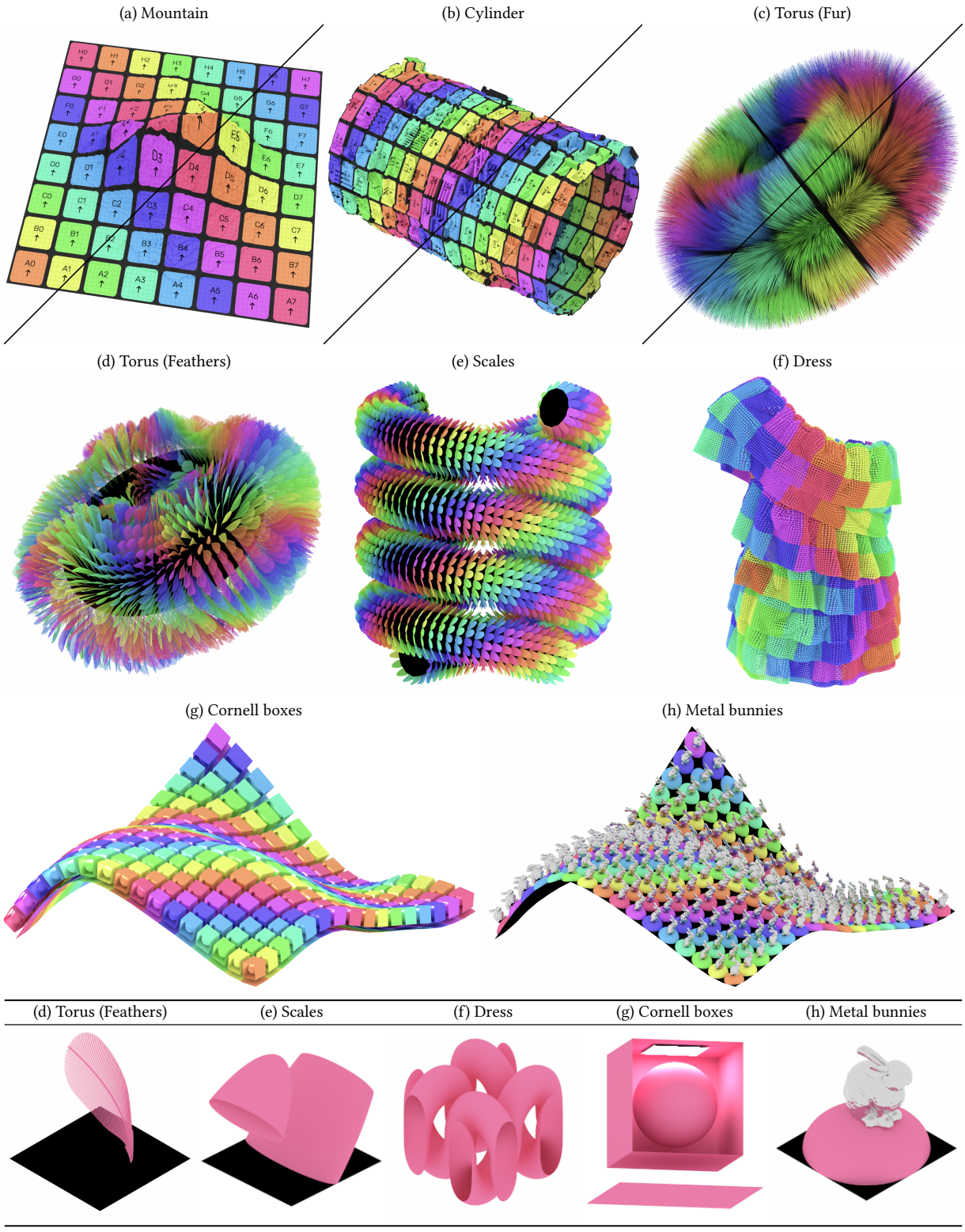

Figure 7: Test scenes. (a)-(c) are displacement-mapped objects. The top left of each is rendered using pre-tessellated geometry, and the bottom right is rendered using our method. (d)-(h) are rendered using our shell mapping method. See Table 2 for the statistics. The images at the bottom are the objects instanced in the shell.

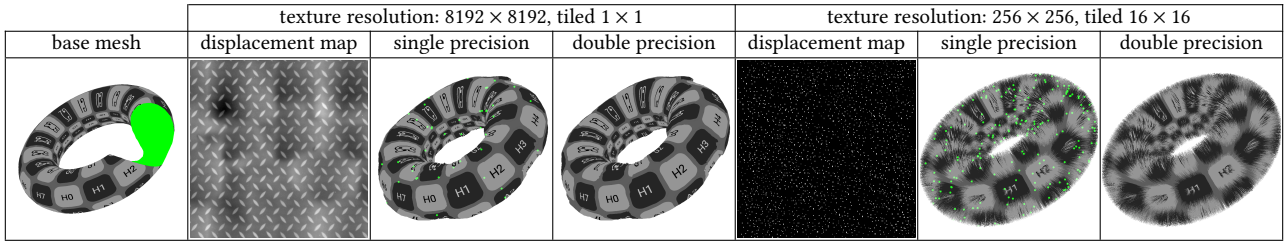

Figure 8: A torus rendered with two different displacement maps. The back surface of the base mesh is bright green luminescent material. The cutout is for explanation purposes only. A small kernel Gaussian filter is applied to the rendered images to highlight where rays slipped through the object. The use of double precision reduces the error. The resolution of each rendered image is 1024×1024, and 1 ray is cast per pixel.

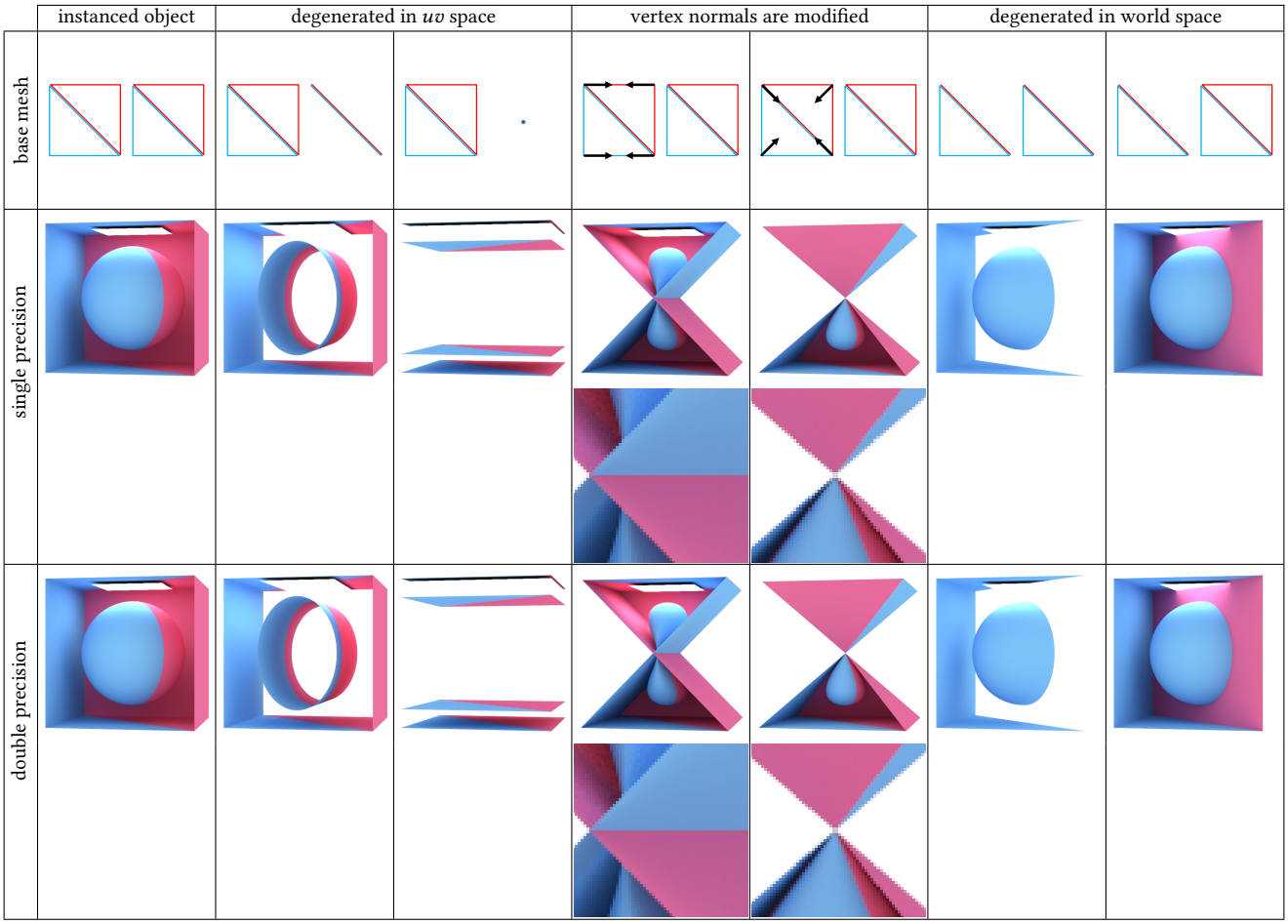

Figure 9: A Cornell box-like geometry defined in texture space is instanced in the shell of a rectangular base mesh consisting of two triangles. The 2D drawings in the first row are the base mesh viewed from the top and the base mesh in uv space, and the black arrows indicate the direction of the vertex normals. In the second and third columns, the base mesh is degenerate in uv space. In the fourth and fifth columns, the vertex normals are oriented so that the prisms degenerate to a line or point at a particular h value. In the second last column, the red base triangle is degenerated into a line in both world and uv space. It disappears as expected as rays and the prism do not intersect. In the last column, the red base triangle is degenerated to a line in world space but not in uv space. This is an deliberately created situation that would not happen in practice. All the test cases are rendered without artifacts with both single and double precision. The resolution of each rendered image is 512×512.