# Tessellation-Free Displacement Mapping for Ray Tracing

THEO THONAT, Adobe, France
FRANCOIS BEAUNE, Adobe, France
XIN SUN, Adobe, USA
NATHAN CARR, Adobe, USA
TAMY BOUBEKEUR, Adobe, France

Fig. 1. Our displacement BVH injects high frequencies onto a 3D surface mesh, here in a GPU path tracer. The whole scene geometry is represented using only 460 MB in GPU memory, with a superior rendering quality than what could be obtained using all the available memory on tessellation. Our method can reuse the same acceleration structure when mapping a displacement map onto different base surfaces, even with different displacement parameters, as can be seen with the two spheres on the right.

Displacement mapping is a powerful mechanism for adding fine to medium geometric details over a 3D surface using a 2D map encoding them. While GPU rasterization supports it through the hardware tessellation unit, ray tracing surface meshes textured with high quality displacement requires a significant amount of memory. More precisely, the input surface needs to be pre-tessellated at the displacement map resolution before being enriched with its mandatory acceleration data structure. Consequently, designing displacement maps interactively while enjoying a full physically-based rendering is often impossible, as simply tiling multiple times the map quickly saturates the graphics memory. In this work we introduce a new tessellation-free displacement mapping approach for ray tracing. Our key insight is to decouple the displacement from its base domain by mapping a displacement-specific acceleration structures directly on the mesh. As a result, our method shows low memory footprint and fast high resolution displacement rendering, making interactive displacement editing possible.

CCS Concepts: • **Computing methodologies** → **Rendering**; **Ray tracing**; **Texturing**; *Mesh geometry models.*

Additional Key Words and Phrases: Ray tracing, displacement mapping, affine arithmetic

**ACM Reference Format:**
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 (December 2021), 16 pages. https://doi.org/10.1145/3478513.3480535

Authors' addresses: Theo Thonat, Adobe, France, thonat@adobe.com; Francois Beaune, Adobe, France, beaune@adobe.com; Xin Sun, Adobe, USA, xinsun@adobe.com; Nathan Carr, Adobe, USA, ncarr@adobe.com; Tamy Boubekeur, Adobe, France, boubek@adobe.com.


0730-0301/2021/12-ART282 $15.00
https://doi.org/10.1145/3478513.3480535

## 1 INTRODUCTION

Displacement mapping [Cook 1984] has proven to be an intuitive approach for artists to add geometric details to 3D surfaces. The strong correlation between material and shape makes storing and editing of displacement information in image form alongside textures quite natural. Both games and film production have leveraged this representation; however, the use of displacement mapping in interactive and real-time ray tracing applications remains an open challenge. Indeed, this often induces an actual tessellation of the base surface domain, performed at the granularity of the displacement map, prior to the construction of static acceleration structures. In this work we present a novel algorithm to interactively ray-trace displacement maps without tessellation. Specifically our approach retains displacement map information in its native image form during ray tracing. By doing so, our method supports instancing and tiling, resulting in significant memory savings over pre-tessellation. Furthermore, our method supports continuous level-of-detail which is critical for reducing geometric aliasing effects encountered during rendering.

Interactive applications have long leveraged rasterization based approaches to synthesize images from 3D scenes. GPU hardware support has added tessellation shaders, enabling techniques to render efficiently displacement mapping and rich geometric detail in real-time [Nießner and Loop 2013]. In contrast, handling displacement mapping in the context of interactive (GPU supported) ray tracing is far more complicated. This is primarily due to the fact that ray-tracers must efficiently support random access spatial queries for geometric information. Specifically, large portions of the scene must be accessible and acceleration structures maintained to support the fundamental ray query operation. The naïve approach of pre-tessellating displacement maps into meshes does not scale. Both the available memory, and computational cost to rebuild finely tessellated meshes and their related acceleration data-structures puts a strict cap on the level of fidelity achievable. For this reason explorations have been made into how to make ray tracing more streamable using caching or ray reordering to support displacement mapping [Christensen et al. 2003; Pharr and Hanrahan 1996; Pharr et al. 1997]. These approaches have mostly been used for offline rendering tasks due to their overhead.

In our work, we focus on developing a practical algorithm for ray tracing displacement maps that can be used in interactive global illumination engines and on arbitrary base surface meshes. We demonstrate our method running in the unbiased GPU Monte Carlo path tracer of the *Adobe Mercury Rendering Engine*. Specifically, our work focuses on allowing displacement information to be handled similarly to texture maps; freely allowing tiling, scaling, instancing, and pre-filtering at interactive rates. We do this while significantly reducing memory consumption, rendering scenes that would not fit into memory using pre-tessellation approaches. To achieve this we propose a novel approach which dynamically constructs acceleration structures tailored for the displacement and "mapped" on the base surface on the fly during ray tracing without needing to cache high-resolution geometry. In summary, our contributions include:

- A method to efficiently perform direct ray tracing of displacement maps over general, non flat surface meshes.
- A bounding volume hierarchy defined in texture space, exhibiting minimal coupling with the base mesh onto which it is mapped.
- A stack-less, pointer-less traversal mechanism running in UV space which skips empty space and optimizes for the bounding volume test order.
- A flexible framework making possible to use customized intersection tests over the canonical base triangle mesh/displacement map input pair, yielding a wide spectrum of geometric fidelity levels with continuous level-of-details.
- Natural level-of-detail (LoD) support that can reduce geometric aliasing effects during rendering and improve performance.

As a consequence, our approach enables interactive rendering of dynamic displacement maps without resorting to high-resolution tessellation. The displacement content as well as its mapping/tiling can be interactively edited, up to very high resolutions, which makes our method particularly suited for modern parametric, non-destructive modeling workflows.

## 2 PREVIOUS WORK

### 2.1 Ray-traced displacement maps

The search for efficient algorithms for ray-tracing geometrically complex scenes has been explored in the context of offline rendering, including numerous extensions to the Reyes architecture. Both caching and ray-reordering have been shown to be effective algorithms [Christensen et al. 2003]. The idea of doing lazy fast building of accelerations structures was explored by Hunt et al. [2007]. Hanika et al. [2010] developed a two-level acceleration scheme building a top level structure over conservative patch bounding volumes. On-demand tessellation and acceleration structure building is performed for rays that hit leaf nodes of this top level structure, and ray-reordering is employed to maximize re-use. In contrast, we use an implicit acceleration structure over the patch, generating the micro structures only at the scale of the displacement map texels.

Djeu et al. [2011] developed the Razor system which enables crack-free multiresolution geometry, allowing each ray to independently choose its appropriate geometric resolution. They use lazy evaluation and caching of tessellations at discrete levels of detail, that are interpolated during intersection to generate fractional LoD. On the contrary, we intersect the displaced surface directly at the fractional LoD, with the displaced surface never being generated explicitly, except during the traversal ultimate step at the scale of a single texel. Hou et al. [2010] expanded the BVH leaf node w.r.t micro-polygon's high dimensional behavior, including motion blur and depth-of-field. This method does not bound w.r.t. base polygons. Our work is orthogonal, processing the relationship between a base polygon and its high-resolution displacement content.

### 2.2 Real-time displacement mapping

In the context of rasterization, several methods have been proposed to render surface details in real time without tessellation. Indeed, the simplest form of displacement mapping is *bump mapping* [Blinn 1978] which uses a normal map to take in account local surface

variations during shading. However, as the surface is not modified, self-shadowing and silhouettes cannot be represented.

*Parallax mapping* [Kaneko et al. 2001; Tatarchuk 2006] uses an height map to find an approximate intersection with the meso-surface from the height information at the macro-surface. Iterative methods have been developed to explore the height field by sampling the height function along the ray using *ray-marching*. They often combine a first search using fixed steps to find a pair of points above and below the surface, followed by a refinement process such as the secant method [Yerex and Jägersand 2004] or a binary search [Policarpo et al. 2005]. In both cases, there is no guarantee that the returned intersection is the closest one. Lee et al. [2009] further improved the linear search of relief mapping using a minmax texture.

Height maps can be used to encode empty space near the surface. This encoding gives a conservative estimation for the next ray-marching step and is generally a map that stores parameters of a simple parametric shape, for example a sphere [Donnelly 2005; Hart 1996] or a cone [Dummer 2006]. In the context of ray-marching against the displacement, empty space encoding can be relaxed to define regions where it is safe to switch to a fast refinement method, for example using *relaxed cone stepping* [Policarpo and Oliveira 2007]. However, computing encoding maps from displacement maps remains an expensive pre-process, which takes several minutes with the most recent methods [Baboud et al. 2011].

The method of Wang et al. [2003] pre-computes intersection with the meso-surface from a dense sampling of query rays and uses Singular Value Decomposition based compression to deal with the amount of generated data in real time. While its extension [Wang et al. 2004] is able to handle a variety of base surfaces and vector displacement, the memory-quality trade-off makes it impractical for high quality displacement from arbitrary input viewpoint.

While parallax mapping and relief mapping have been demonstrated in real-time applications, special handling is required to enable additional effects like silhouette edges, shadows, object and interpenetrations [Chen and Chang 2008; Dachsbacher and Tatarchuk 2007; Oliveira and Policarpo 2005]. Our approach ensures a globally consistent geometry and can be naturally integrated into any path-tracing supporting the full range of effects (see Figure 14). A more general overview on methods to render meso-surface in real time on the GPU can be found in the report of Szirmay-Kalos et al. [2009].

## 2.3 Implicit acceleration structures for raytracing

The idea of having an implicit data structure to overcome the memory burden produced by a very detailed geometry is not new. Similarly as our work, Heidrich and Seidel [1998] use affine arithmetic to estimate displacement bounds and base surface bounds in *uv* space. Their work is focused on procedural displacement shaders and contrary to our method, does not take in account the curvature of the base mesh. The work of Smits et al. [2000] also allows direct ray tracing of displaced polygonal meshes without tessellation with a different tradeoff as our method. Their traversal is based on a ray marching prisms in barycentric space, allowing their method to cover exactly the base domain. However, their displacement bounds are constant per triangle, which leads to loose estimations for high resolution or high amplitude displacement. Carr et al. [2006] showed how to efficiently ray-trace scenes comprised as geometry images. They build an implicit quad tree bounding volume hierarchy by decomposing the model in *uv* space. In contrast, our method decouples base level geometry (i.e. triangle mesh information) from displacement information.

Oh et al. [2006] use a maximum displacement mipmap as a hierarchical data structure to intersect height fields. However, their traversal suffers from limitations such as missing intersections, as it is unable to ascend the hierarchy. The work of Tevs et al. [2008] overcome these limitations by using maximum displacement mipmaps as an implicit bounding volume hierarchy, in the special case of height field rendering. Our method is a generalization of their method to any polygon base mesh.

## 2.4 Smooth surfaces raytracing

A number of methods have been proposed for ray tracing efficiently smooth surfaces both without or with displacement data. For instance,Benthin et al. [2015] cache local tessellation for ray tracing subdivision surfaces. Moreton [2001] presented a hardware algorithm for watertight tessellation of polynomial surfaces. Nießner et al. [2012] developed a high performance GPU approach for rendering Catmull-Clark Subdivision Surfaces. Selgrad et al. [2016] define a priori BVHs for smooth parametric patches, which avoid pretessellating them. Lier et al. [2018] proposed a compressed scheme, using localized compact approximation of the displaced geometry to speed-up rendering.

Regarding data structure, Munkberg et al. [2010] build an efficient bounding structure of displaced Bézier patches, using a min-max hierarchy to construct tight oriented bounding volumes. Lu et al. [2009] propose to decompose a mesh into height fields in a octree, and ray-trace against the height fields.

Wang et al. [2000] detect and analyze displacement features to reproduce them faithfully. Moule and McCool [2002] propose an on-the-fly, crack-free solution to displacement mapping which uses local triangle information to provide a view-dependent tessellation.

## 2.5 Shell mapping

Porumbescu et al. [2005] creates a thin intersectable volumetric region around the surface by extruding the mesh outward in the normal direction. Intersection with this volume is performed and in the case of displacement mapping, the ray can be transformed into the local texture space to find an appropriate intersection using e.g., local height field ray-tracing. Unfortunately, the mapping from world to texture space is non-linear, and thus requires tracing curved ray paths to avoid the height field from buckling [Jeschke et al. 2007]. As with many shell based methods [Hirche et al. 2004], issues can arise if the mapping is not bijective e.g, when the extruded prisms invert and self intersect. This limits the maximum displacements to the local radius of mesh curvature unless care is taken. Our algorithm is always guaranteed to construct a valid bounding box hierarchy and is robust to these conditions.

We refer the reader to the survey on displacement mapping provided by Szirmay-Kalos and Umenhoffer [2008] which covers a number of high performance methods, including GPU-based bump

Table 1. Terminology

| Symbol | Type | Comments |
|---|---|---|
| $\mathbf{P}$ | vec3 | Base surface position |
| $\mathbf{N}$ | vec3 | Base surface interpolated normal |
| $\hat{\mathbf{N}}$ | vec3 | Normalized interpolated normal |
| $h$ | float | Sampling of a $W \times H$ displacement map |
| $uv$ | vec2 | Texture coordinates $u$ and $v$ |
| $i, j$ | ivec2 | Texel integer coordinates |
| $k$ | int | Texel mipmap level |
| $\Omega^k_{i,j}$ | vec2[2] | $uv$ domain corresponding to texel $(i, j, k)$ |
| $M$ | vec2 | Minmax mipmap relative to a sampling $h$ |
| $\underline{x}, \overline{x}$ | | Min and max of the quantity $x$ |
| $[a, b]$ | | Interval, or AABB for vector quantities |
| $[x]$ | | Affine form, also written as $[x_c, x_u, x_v, x_K]$ |

mapping, parallax mapping, relief mapping, horizon mapping, cone stepping, local ray tracing, pyramidal and view-dependent displacement mapping methods, as well as their numerous variations, which are reviewed extensively, providing also implementation details of the shader programs.

## 3 OVERVIEW

Our method provides a direct ray tracing operator with low memory footprint for surfaces enriched with displacement maps. The key idea is to define a displacement-centric acceleration structure that can be mapped onto polygon meshes similarly as texture mapping. The actual bounding volume hierarchy for the displaced surface is never stored in memory and is generated on the fly on a per-ray basis. Being free of expensive base domain pre-computation, e.g. tessellation, our method makes possible interactive editing of the displacement content.

An overview of our method is given in Figure 2. We show in section 4 the construction of the displacement acceleration structure – that we call *Displacement Bounding Volume Hierarchy* or *D-BVH* – and an algorithm to traverse it to compute intersections with the displaced surface. The D-BVH is quite general and allows to vary several components that we detail in section 5, such as the type of displacement or the sampling of the displacement map. We give in section 6 details about our practical implementation for interactive GPU ray tracing. Finally, we show in section 7 results in terms of appearance, memory footprint and performances, as well as comparisons with previous methods, and an ablation study of the method optimizations. Table 1 gives the terminology for the main symbols encountered in the paper.

## 4 DISPLACEMENT BVH

We present in this section a method to compute and use the *D-BVH*. The D-BVH maps pre-computation of displacement bounds in texture space to compute 3D bounding volumes during ray tracing. It hybridizes a 2D hierarchical space division structure at the texture level and an implicit 3D bounding volume hierarchy for the displaced surface.

### 4.1 Displaced surface

We define a displaced surface for any texture coordinate $(u, v)$ as:

$$\mathbf{S}(u, v) = \mathbf{P}(u, v) + h(u, v)\hat{\mathbf{N}}(u, v) \tag{1}$$

where $\mathbf{P}$ and $\hat{\mathbf{N}}$ are respectively position and unit normal of the base surface, and $h$ is a sampling of the displacement map. For base surfaces we consider, $\mathbf{P}$ and $\mathbf{N}$ are defined per base primitive using linear interpolation of the vertices properties, and we define the complete displaced surface as the union of all displaced surfaces generated by each base primitive. Therefore, we will always consider ray tracing the displaced surface restricted to a single base primitive. We show in section 5 an example where the base surface is a triangle mesh, and detail in section 6 a practical implementation for a displaced object representing all its displaced triangles. From Equation 1, we see that estimating a bounding volume of the displaced surface for a $uv$ domain means computing bounds for the base position, the base normal, and the displacement. Our D-BVH takes advantage of displacement being independent from the base surface embedding and store bounds for all $uv$ domains considered during traversal. We first describe the D-BVH construction. We then describe the *D-BVH* traversal procedure, and how we use it in combination with *affine arithmetic* to estimate 3D bounds for the displaced surface. Finally we explain how integer and continuous level-of-detail are supported by the *D-BVH*.

### 4.2 Construction

The work of Tevs et al. [2008] uses a maximum mipmap as an implicit acceleration structure to raymarch displacement maps applied to planar base surfaces. In their setup, five planes can be computed on the fly for each maximum mipmap texel to bound the displaced surface, the base surface acting as an implicit sixth plane to complete the bounding box. Inspired by this work, we use a minmax mipmap as an acceleration structure for the displacement, storing both a conservative minimum and maximum value of displacement over the $uv$-domain corresponding to each texel. More specifically, given a sampling $h$ of a displacement map with maximum resolution $W \times H$ and $K$ mip levels, we define its minmax mipmap $M^k_{ij}$ for $0 \leq k < K$, $0 \leq i < W2^{-k}$, and $0 \leq j < H2^{-k}$ as:

$$M^k_{i,j} = \left[\underline{M}^k_{i,j}, \overline{M}^k_{i,j}\right] = \underset{(u,v)\in\Omega^k_{i,j}}{\text{minmax}}\ h(u, v) \tag{2}$$

where $\Omega^k_{ij}$ is the $uv$ domain corresponding to a texel, defined by $\left(i2^k \leq W \cdot u \leq (i+1)2^k\right) \wedge \left(j2^k \leq H \cdot v \leq (j+1)2^k\right)$ for any integers $i, j, k$. This minmax mipmap is effectively a 2-channel texture with the same dimensions and same number of mipmap levels as the input displacement map, and is independent from the base surface on which the displacement is mapped on.

In practice, the minmax mipmap is computed starting from level $k = 0$ using sampling-dependant formulas, and with the following recursive formula for the other levels:

$$M^k_{i,j} = \underset{s/2\,=\,i,\ t/2\,=\,j}{\text{minmax}}\ M^{k-1}_{s,t} \tag{3}$$

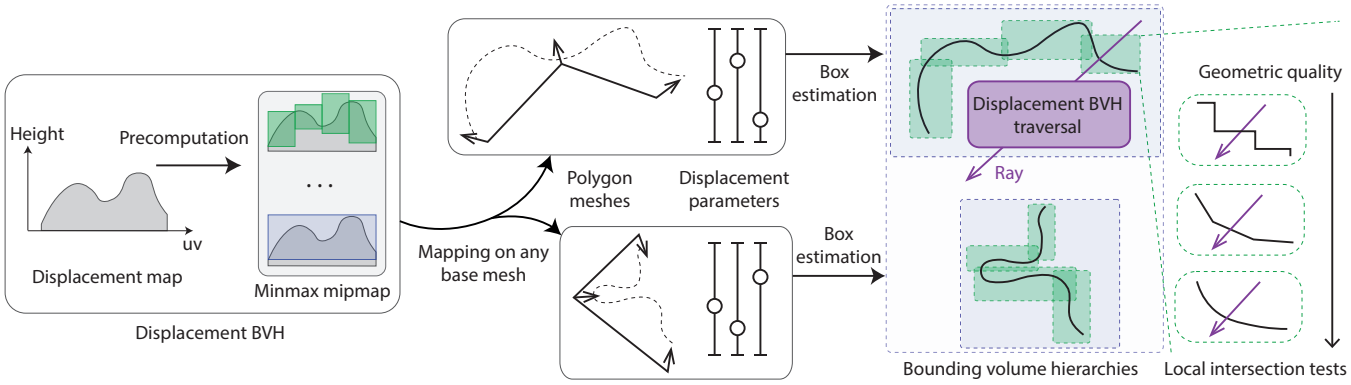


Fig. 2. **Method overview:** we compute an acceleration structure (D-BVH) at the displacement level using a minmax mipmap. This structure is then mapped, with low memory overhead, onto any polygon mesh to provide direct ray tracing of surfaces with displacement. The actual displaced surface is never stored in memory and its bounding volumes are computed on the fly on a per-ray basis. Displacement parameters, such as the $uv$ mapping or the displaced surface geometric quality, can be modified at any time with minor additional computations.

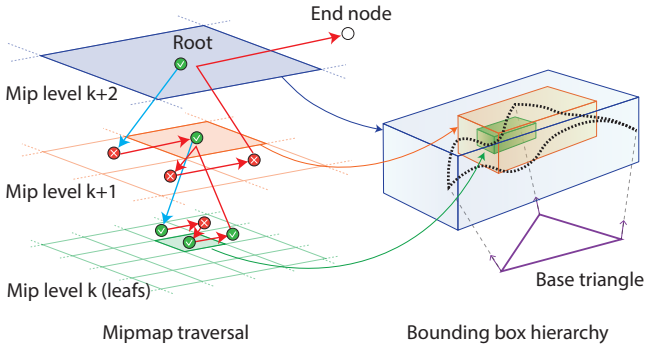


Fig. 3. **D-BVH traversal:** The minmax mipmap is traversed (left) to compute a hierarchy of bounding volumes generated on the fly (right). Light blue arrows represent the *down* operator and red arrows the *next* operator as defined in algorithm 1. A traversal starts from the root and considers texels until the end texel is reached. Green check-marks and red-crossings indicate whether the ray has intersected or not the box generated by the texel. In case of intersection, the traversal goes down to the next mip level, unless a leaf was reached and a local intersection test with the displaced surface is performed.

## 4.3 Traversal

Given a ray and a base triangle, our D-BVH traversal visits all texels, at the finest mip level, which could generate a displaced surface that intersects the ray. We use the hierarchical nature of the mipmap to perform early discard of large $uv$ domains and accelerate the traversal.

The mipmap defines an implicit graph structure, where nodes are simply texels (2 integer coordinates and 1 integer mip level), and where children of a node are simply its 4 corresponding texels in the mip level below. This implicit graph is not restricted to texels corresponding to actual displacement data, but also contains any signed integer mip layers and texel coordinates. This extension is useful to handle texture coordinates outside the unit square, which often happens when tiling textures.

Starting the traversal from the root and assuming we have a criterion to identify leaf texels, our traversal is simply a depth first search in a subtree of this infinite implicit graph, as shown in Figure 3. As the graph is implicit, the traversal is stack-less and pointer-less. We define leafs as texels whose mip levels are below a given target mip level and for which the displaced surface has a closed-form that can be directly intersected. Examples of such closed-forms with different geometric qualities are given in section 5. At each traversed texel, a 3D axis-aligned bounding box is computed by combining bounds from the D-BVH and information from the base triangle. The box is then intersected with the query ray using the slab method [Williams et al. 2005]. This intersection determines if we either go down in the hierarchy or just skip the texel. If we are at a leaf, we perform the actual intersection test with the displaced surface, and update the intersection information accordingly. Pseuso-code for the traversal procedure is shown in algorithm 1. Details about the root choice, the traversal order or texel early discard are given in section 6.

## 4.4 Level of detail

We discuss in this section how our *D-BVH* handles different integer and continuous levels of detail using the same hierarchy.

*Integer level-of-detail.* As noted by Carr et al. [2006] and Tevs et al. [2008], minmax mipmap traversals handle integer level-of-detail (LoD) almost for free by simply skipping texels below the target level of interest. However, while the kernel support of the minmax-mipmapping has the same size as standard box-mipmapping, the height sampling function might fetch data from outside this support, even with simple bilinear filtering, as seen in Figure 4. As a consequence, the level $k$ of the minmax mipmap as defined in subsection 4.2 is not always a conservative estimate of the level $k$ of the box-mipmapped input displacement. Therefore, in order to represent all displacement LoDs in the same hierarchy, the recursion formula from Equation 3 has to be slightly modified to also consider the height sampling $h^k(u, v)$ at mipmap level $k$:

**Algorithm 1:** Mipmap traversal for ray tracing
The down, up, and next functions describe the texels traversal order and modify in place their argument. The box and local_intersection functions – respectively computing the bounding box from a texel and performing the actual intersection with the displaced surface – is detailed in section 5. The root function, computing the traversal starting texel, and the outside function, performing early texel discard, are described in section 6. The miss function is based on a ray-AABB intersection routine.

```
Function down(texel):
    --texel.LoD
    texel.ij *= 2
Function up(texel):
    ++texel.LoD
    texel.ij /= 2
Function next(texel):
    while true do
        switch 2*(texel.i % 2) + texel.j % 2 do
            case 1 do
                --texel.j
                ++texel
                return
            case 3 do
                up(texel)
                continue
            else
                ++texel.j
                return
Function traversal(ray, triangle, target LoD):
    intersection = empty
    texel, endtexel = root(triangle)
    next( endtexel)
    while texel ≠ endtexel do
        if outside(texel, triangle) or miss (ray,
          box(texel)) then
            next(texel)
        else if texel.LoD ≤ targetLoD then
            if hit = local_intersection(texel, ray) then
                update intersection from hit
            next(texel)
        else
            down(texel)
    return intersection
```

$$M^k_{i,j} = \text{minmax}\left(\min_{s/2=i,\, t/2=j} M^{k-1}_{s,t}, \min_{\Omega^k_{i,j}} h^k(u,v)\right) \quad (4)$$

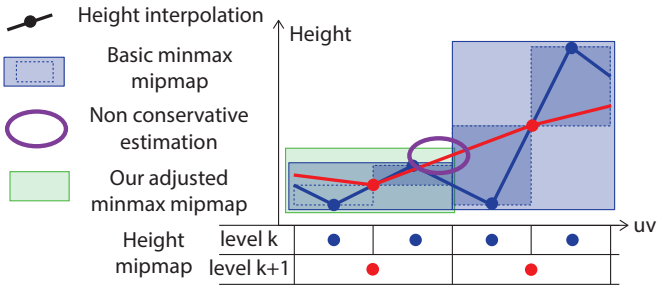


Fig. 4. **Minmax mipmap LoD capabilities:** The linearly interpolated height at LoD $k$ (bold blue line) is conservatively estimated by the minmax mipmap at level $k$ (dark blue boxes). The minmax mipmap at level $k+1$ (light blue boxes), computed using standard minmax mipmapping from Equation 3, is indeed a conservative estimation of the interpolated height at LoD $k$. However, as highlighted in purple, it is not a conservative estimate of the linearly interpolated height at LoD $k+1$ (bold red line). By using the adjusted minmax mipmapping from Equation 4, we obtain correct bounds for all displacement mip levels (light green box),

*Continuous level-of-detail.* As seen from Equation 1, the displaced surface is linear with respect to the height. Therefore, the displaced surface at a linearly blended height from two consecutive integer LoDs is exactly the linear blend between the displaced surfaces at those LoDs. More precisely, given a non-integer LoD $l$ such that $k < l \leq k+1$, and a blending parameter $\lambda = l - k$, we define the displaced surface at $l$ as:

$$\begin{aligned}\mathbf{S}^l(u,v) &= \mathbf{P}(u,v) + h^l(u,v)\hat{\mathbf{N}}(u,v)\\ &= \mathbf{P} + \left((1-\lambda)h^k + \lambda h^{k+1}\right)\hat{\mathbf{N}}\\ &= (1-\lambda)\left(\mathbf{P} + h^k\hat{\mathbf{N}}\right) + \lambda\left(\mathbf{P} + h^{k+1}\hat{\mathbf{N}}\right)\\ &= (1-\lambda)\mathbf{S}^k(u,v) + \lambda\mathbf{S}^{k+1}(u,v)\end{aligned} \quad (5)$$

The minmax mipmap directly provides bounds for the LoD blended height. For the upper bound, we have:

$$\begin{aligned}\max_{\Omega^k_{i,j}} h^l &\leq \max\left(\max_{\Omega^k_{i,j}} h^k, \max_{\Omega^k_{i,j}} h^{k+1}\right) && \text{Convex combination.}\\ &\leq \max\left(\overline{M}^k_{i,j}, \max_{\Omega^{k+1}_{i/2,j/2}} h^{k+1}\right) && \text{Since } \Omega^k_{i,j} \subset \Omega^{k+1}_{i/2,j/2}.\\ &\leq \overline{M}^{k+1}_{i/2,j/2} && \text{Using } \textit{Equation } 4.\end{aligned} \quad (6)$$

And similarly, we have $\min_{\Omega^k_{i,j}} h^l \geq \underline{M}^{k+1}_{i/2,j/2}$ for the lower bound. Thus, the traversal described in algorithm 1 is able to intersect the displaced surface at non-integer LoD by simply modifying Equation 11 so texel $M^{k+1}_{i/2,j/2}$ is fetched instead of texel $M^k_{i,j}$.

For local intersections involving continuous height sampling, the displaced surface is continuous with respect to the LoD, as shown in Figure 5. For local intersections involving discrete height sampling, the displaced surface is not continuous with respect to the LoD as the sampling pattern changes when $l$ crosses $k$. In practice however,


Fig. 5. **Continuous geometric LoD:** (a) Displaced object rendered at maximum LoD. (b) Our D-BVH is able to render different non-integer LoD for each ray, driven by the screen-space $y$ position of the primary ray. (c) LoD influence on the number of traversed nodes.

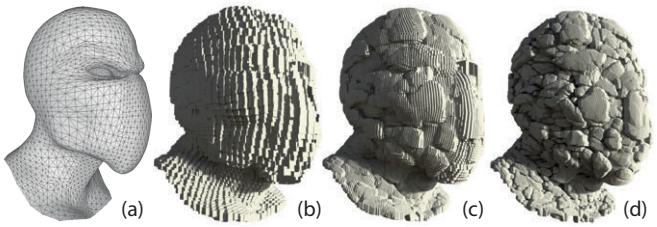


Fig. 6. **D-BVH bounding boxes:** By mapping a displaced BVH on a base mesh (a), our method computes on-the-fly per-ray bounding boxes of the displaced surface. (b-d) Rendering of the union of those boxes for different mip levels, here in object space.

the non-integer LoD still provides visually appealing morphing between integer LoDs.

### 4.5 Conservative box estimation

During traversal, we compute on the fly, for each traversed mipmap texel, a conservative 3D axis-aligned bounding box (AABB) of the displaced surface. As our base surface is not always flat, we need to also estimate the deviation induced by the base triangle interpolated normal. Similarly to Heidrich and Seidel [1998], we use *affine arithmetic* [De Figueiredo and Stolfi 2004] to estimate and combine positions on the base surface, displacement values and interpolated unit normals. Affine arithmetic is an improvement over other simpler estimation arithmetic such as *interval arithmetic* as it allows to take into account correlation between the estimated variables meaning that approximations can cancel themselves out instead of always being accumulated. In affine arithmetic, quantities are affine forms, which are convex sets with central symmetry, so both the input texel $uv$ and the output bounding box can be represented by a single affine form.

*On the fly computation.* The minmax mipmap can be interpreted as a pre-computation of height intervals for every texel $uv$ domain. Therefore the affine form associated to the height relative to a texel can be directly retrieved from a single minmax mipmap fetch. On the other hand, affine forms for the interpolated base position and unit normal can be directly computed from the $uv$ affine form. Finally, the affine form for the displaced surface over the $uv$ domain can then be computed using Equation 1, from which we extract an axis aligned bounding box as shown in Figure 6. More details on affine arithmetic computations and formula can be found in Appendix A.

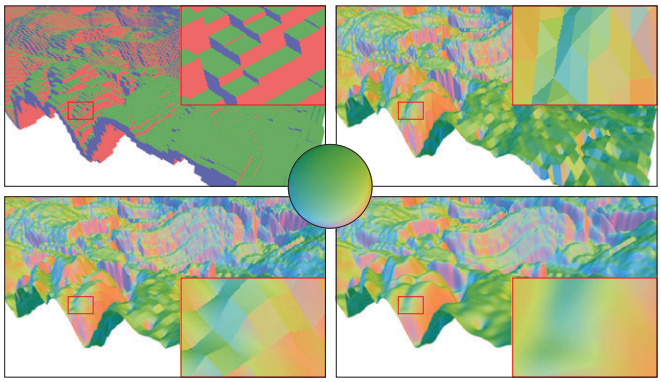

Fig. 7. **Local intersections:** Geometric normal visualization for different types of local intersection: leaf boxes (top left), local triangulation with two triangles per texel (top right), bilinear height sampling (bottom left), and B-spline height sampling (bottom right). The upper-half hemisphere of directions is shown at the center.

## 5 RAY TRACING

Our base surface model is a triangle mesh, where position and normal are obtained by linearly interpolating the vertex attributes and for which we can explicitly compute Equation 1 for each individual triangle. Given vertex positions $p_i$, normals $n_i$ and texture coordinates $u_i$ and $v_i$ for the triangle three vertices, we have:

$$\mathcal{T}_{uv}^{b} = \begin{pmatrix} u_1 & u_2 & u_3 \\ v_1 & v_2 & v_3 \\ 1 & 1 & 1 \end{pmatrix}^{-1} \tag{7}$$

$$\mathbf{P}(u,v) = \begin{pmatrix} | & | & | \\ p_1 & p_2 & p_3 \\ | & | & | \end{pmatrix} \mathcal{T}_{uv}^{b} \begin{pmatrix} u \\ v \\ 1 \end{pmatrix} \tag{8}$$

$$\mathbf{N}(u,v) = \begin{pmatrix} | & | & | \\ n_1 & n_2 & n_3 \\ | & | & | \end{pmatrix} \mathcal{T}_{uv}^{b} \begin{pmatrix} u \\ v \\ 1 \end{pmatrix} \tag{9}$$

where $\mathcal{T}_{uv}^{b}$ is a $3 \times 3$ matrix which computes the barycentric coordinates of a point given its $uv$ texture coordinates. We use scalar displacement maps, equipped with a sampling operator, and over which we build our D-BVH. During traversal, we compute positions and normals in the local $uv$-aligned tangent space. This local space coincides with the base triangle when $z = 0$. Its first two axes are aligned with the $u$ and $v$ axes and its third axis is the base triangle geometric normal, as seen in Figure 9. We use this local space instead of object space to minimize bounding boxes' volumes as they are likely to be aligned with the displaced surface.

### 5.1 Local intersection tests

Our method supports a variety of intersection tests which are illustrated in Figure 7. The micro-primitives considered during traversal for each intersection test are illustrated in Figure 8 .

*Box.* The simplest local intersection test that can be performed is to just return the intersection with the leaf AABB. As the box and its intersection with the query ray are already computed as part of

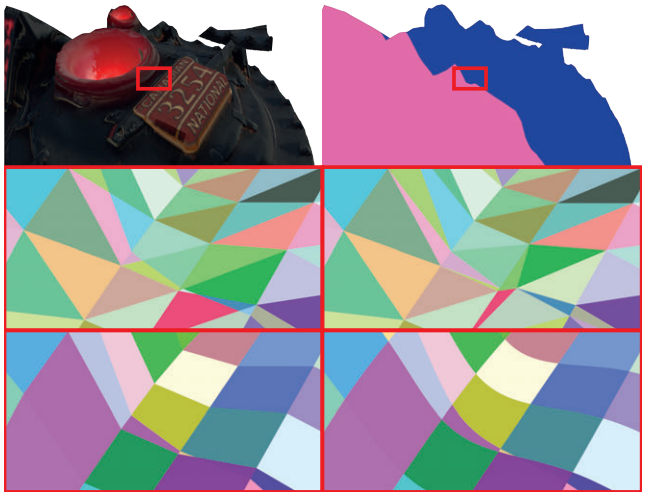

Fig. 8. **Leaves domain and base primitives boundary.** Top row: A rendering of a displaced mesh (left), made of two base triangles (right). Next rows focus on the boundary between the two displaced base triangles in the red rectangle closeup. Middle row: Micro-triangles considered during traversal for the two-triangles-per-leaf solution (left), and with texel clipping (right). Note that in the two-triangles approach, all micro-triangles have the same size in $uv$ space, but since they overlap at the base triangles boundary, it appears otherwise in the rendering. Bottom row: For continuous height sampling, micro-primitives are simply bilinear (left) or bicubic (right) patches, corresponding to $uv$ domains between four texel centers.

the traversal, this leaf-box intersection has no overhead. Since the resulting displaced surface has piece-wise constant normals, this intersection is useful in scenario where precision is not critical and where no shading information is needed, such as shadow rays.

*Local triangulation.* One way to perform a global pre-tessellation of a displaced mesh would be to compute one vertex per texel corner at the lowest mip level, with two triangles per border-aligned texel. Our *local triangulation* intersection mode corresponds to on-the-fly, per-ray alternative to such a pre-tessellation. At the leaf scale, our $uv$ domain is an center-aligned texel so it overlaps four border-aligned texel (see Figure 8). Once a leaf is reached, we generate on the fly, for each of those border-aligned texels, two ephemeral micro triangles that are analytically intersected with the ray using the the method of Möller and Trumbore [1997]. The resulting surface is continuous, except possibly at the base primitive boundaries, and has sufficient shading quality for a medium range of distances along the ray, or when the displacement map exhibits a high resolution.

*Bilinear and B-spline height sampling.* Here we consider the parametric displaced surface defined by Equation 1. At the leaf scale, as the current texel and its neighbors are known, the height sampling has a closed-form formula for both bilinear and B-spline interpolation schemes. There is unfortunately no analytical formula for the intersection between the ray and the local surface when the base mesh is not flat. However, as the surface has an explicit parametric formula, we can use iterative optimisation methods such as Newton's to find the intersection, as done for example by Martin et al. [2000]. Implementation details can be found in Appendix B. The B-spline scheme adds over the base surface a $C^1$ surface everywhere and provides a high geometric quality. However it has a higher computational cost than previously mentioned local intersections. On the other hand, the bilinear scheme suffer from $C^1$ discontinuities at texel borders with similar computational costs as the B-spline scheme.

### 5.2 Watertightness considerations.

As seen from Equation 1, the displaced surface is continuous whenever the textures coordinates, the base surface, and the displacement are continuous. The case of surface cracks generated by $uv$ seams is outside of the scope of this work.

We first discuss the scenario where the LoD is constant per displaced surface. For intersection tests based on continuous height sampling such as B-spline interpolation, the displaced surface is watertight by definition for any fixed integer of fractional LoD. Note however that in practice, because of the iterative optimisation scheme, intersections can be missed. For intersection tests based on local triangulation, the two triangle per leaf solution is not watertight as the generated micro triangles from two different base primitives do not have to match at the edge between the two base primitives. The possible cracks are only visible when leaves size is close to the base primitive size in $uv$ space, which rarely happens in our target scenario of high resolution displacement maps mapped over low poly base surfaces. Still, a practical solution to this issue is to clip the leaf texel against the base triangle in $uv$ space, triangulate the resulting polygon, and generate the associated displaced micro triangles.

The choice between these two micro-tessellation schemes depends on the desired quality versus speed tradeoff, as the texel clipping guarantees watertightness at the cost of generating up to five micro triangles instead of only two.

For ray-dependent LoD, the surface is also ray-dependant so the definition of water-tightness is unclear. As seen from Figure 5, for continuous height sampling, the surface appears to be watertight if the function that maps rays to their LoD is also continuous but we lack a formal proof.

## 6 IMPLEMENTATION

We implemented our method on GPU using the Vulkan Ray Tracing extension [Khronos 2020], but it is applicable to any ray tracing system supporting custom geometries, which are geometries defined by their bounding box and an intersection program, such as Embree [Wald et al. 2014].

### 6.1 GPU ray tracing pipeline

The ray tracing pipeline described by the Vulkan Ray Tracing extension is made of three main components. The first one is the control flow for a ray, using different programmable shaders for each stage (see Figure 9, left). The second one is the acceleration structure for a scene. Such a structure has two levels: the Top Level Acceleration Structure (TLAS) representing all the object instances in the scene, and the Bottom Level Acceleration Structures (BLAS), each representing distinct objects. The final component, the *shader*

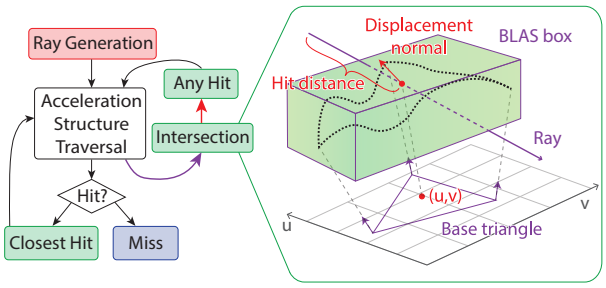


Fig. 9. **Ray tracing pipeline:** Our pipeline is composed of several programmable shaders (colored boxes on the left). Any intersection between the ray and one of the displaced triangle boxes triggers an *intersection shader*, where our traversal is implemented. That means that our method is orthogonal to the shading logic (*closest hit shader*) and to the transparency logic (*any hit shader*). The intersection shader takes as input the base mesh primitive and the ray, and additional data about the displaced object. In case of intersection with the displaced surface, it returns the hit distance, and the intersection point's barycentric coordinates and displacement normal. The displacement normal is used during path tracing.

*binding table* (SBT), is a data structure that links the acceleration structures with the control flow. It assigns, for each instance, the set of programmable shaders to use, as well as instance-specific *inline data* accessible in the shaders.

Our implementation relies on a custom intersection shader for objects with displacement. First, we need to compute the BLAS of each displaced object. Then, for each instance of a displaced object, we have to declare in the SBT the use of our custom intersection shader, as well as data such as displacement parameters.

## 6.2 Displaced object BLAS

As our displaced object is a custom geometry, its BLAS is constructed using a list of AABBs in object space, where each box bounds the displaced surface generated by a single triangle of the base surface. As done in subsection 4.5, we use affine arithmetic to estimate these boxes, the difference being that the base *uv* domain is here a triangle and not a square.

Paiva et al. [2012] noted that affine arithmetic can be performed strictly on a triangle domain by considering a triangle as a union of three overlapping parallelograms. Therefore it is enough to use affine arithmetic separately for each parallelograms and merge their estimated boxes.

Retrieving the displacement bounds with respect to a triangle domain using the D-BVH is possible by adapting the traversal described in algorithm 1 to compute a conservative estimate by visiting all texels overlapping the triangle at the right scale, as detailed in algorithm 2. The computation of the BLAS' boxes using a compute shader takes less than a 0.3 ms for all our test scenes (see Table 3).

## 6.3 Memory layout and interactive displacement

The different data structure layouts are detailed in algorithm 3. A D-BVH is composed of two textures: a displacement map and its minmax mipmap. Mapping a D-BVH onto a base mesh creates a *displaced object*, which in addition to references to the base mesh

**Algorithm 2:** Displacement bounds for BLAS computation. This traversal is adapted from algorithm 1. `outside` and `inside` functions rely on the 2D square-triangle collision test from algorithm 4

```
Function traversal(triangle):
    bounds = ∞ · [1, −1]
    texel, endtexel = root (triangle)
    next(endtexel)
    while texel ≠ endtexel do
        if outside (texel, triangle) then
            next(texel)
        else if inside (texel) or texel.lod == 0 then
            bounds = minmax(bounds, minmax(texel))
            next(texel)
        else
            down(texel)
    return bounds
```

**Algorithm 3:** Displaced object data structures

```
struct DisplacementBVH {
    Texture1f  displacement_mipmap;
    Texture2f  minmax_minmap;
};
struct DisplacementParameters {
    mat3  texture_transform;
    float offset, bias, scaling;
    float target_level_of_detail;
    uint intersection_type;
};
struct DisplacedTriangleData {
    mat4  object_to_triangle_uv_space;
    mat3  uv_to_barycentrics, uv_to_normal;
};
struct DisplacedObject {
    Mesh* base_mesh ;
    DisplacementBVH* displacement_bvh;
    DisplacementParameters parameters;
    BLAS blas;
    Buffer<DisplacedTriangleData>  triangle_buffer;
};
```

and to the D-BVH, also contains displacement-related parameters, a BLAS, and a buffer for per-triangle precomputed data. The parameters are included as inline data in the SBT and can be modified interactively. The triangle buffer is in theory optional, as all its data can be computed on-the-fly in the intersection shader. However, we found that pre-computing per-triangle data is useful for performance and precision reasons. Most notably, the matrix inversion from Equation 7 requires double precision to prevent artifacts.

The displacement parameters allow to modify interactively the appearance of the displaced surface, both locally and globally. We

Table 2. **Interactive displacement modifications:** data structures updates for various displaced surface modifications at runtime. $T$, $P$, and $I$, are respectively the number of triangles of the base mesh, the number of pixels in the displacement map, and the number of instances in the scene. (1) If changing the type of intersection also modifies the height sampling bounds, the minmax mipmap has to be recomputed. A simple solution to avoid re-computations is to pre-compute a minmax mipmap for each type of sampling, or a single conservative minmax mipmap combining all samplings that need to be supported.

| | | Minmax mipmap | BLAS | Shader Binding Table |
|---|---|---|---|---|
| Time complexity | | $O(P)$ | $O(T)$ | $O(I)$ |
| Creating a D-BVH | | X | | |
| Mapping a D-BVH | | | X | X |
| Updating | Texture mapping | | X | X |
| | Displacement scaling | | X | X |
| | Type of intersection | See (1) | | X |
| | Target Level of Detail | | | X |

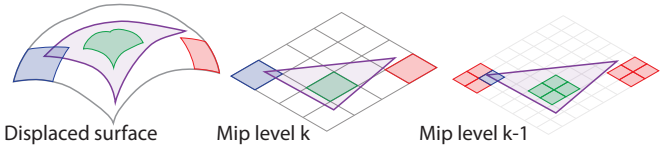


Fig. 10. **Texel skipping:** At mip level $k$ the green and red texels (and all their children) are respectively fully inside and fully outside the $uv$ triangle of interest (in violet). The red texel can be skipped even before computing its 3D bounding box as the displaced surface it generates is fully outside the displaced surface generated by the base triangle. Moreover, there will be no need to check for triangle-texel collision for the green node children. The blue texel is ambiguous and cannot be skipped as its children can be either inside, outside or overlapping the triangle.

show in Table 2 the performance cost of several changes of parameters. Modifying the texture transform changes the way the displacement is mapped onto the base surface. A typical scenario – intensively used in production – is, for a tile-able displacement map, to scale the $uv$ in order to amplify the high frequencies on the final surface. The displacement offset, scaling and bias allow a global linear transformation of the displacement values:

$$h'(u,v) = \text{offset} + \text{scaling} \cdot (h(u,v) - \text{bias})$$

As this operation is linear it has a very low impact on the affine arithmetic computations. While simple, it has several useful subcases such as volume-preserving displacement scaling (using bias = offset = average displacement), which only increases the displacement contrasts without modifying the overall object volume.

### 6.4 Traversal implementation details

*Texel discard.* Since we are considering base surfaces generated from triangle meshes, texels can be traversed while being completely outside the uv domain of the base triangle of interest, as seen in Figure 10. Those texels can be discarded early before computing a bounding box, as any point on the displaced surface generated from those texels would fail the barycentric clipping test. In order to identify these nodes during traversal, we need to determine whether a node overlaps the triangle. We implemented standard polygon collision algorithms such as GJK [Gilbert et al. 1988], pairwise edge intersection, or using separating axis. By considering our special case, where the triangle is fixed and is tested against many squares, we also implemented a custom node-triangle collision test, detailed in Appendix C, providing significant performance improvement (see Table 4).

*Root choice.* In order to start our traversal, we need to find a root texel which contains the base triangle of interest. Assuming the triangle texture coordinates are all positive, one simple solution is to take the smallest texel containing the triangle $uv$ bounding box. However, this strategy performs poorly in case the base triangle overlaps two large texels, as the root would remain the same no matter how small the triangle would be. A more practical solution is to find up to 4 neighboring roots so their union contains the base triangle. This is different from the single root solution as these roots do not have to be children of the same unique node. The traversal with multiple roots performs algorithm 1 once for each individual root.

*Traversal order.* The pseudo-code from algorithm 1 gives an example of procedure to obtain the next texel to traverse thanks to the *next* and *down* functions. However, such fixed order can be inefficient for certain rays, as intersection candidates could be considered in the reverse order of their distance along the ray, thus possibly making the traversal reach leaves several times unnecessarily. We therefore adapt traditional BVH traversals strategy to sort children according to the ray direction in object space (e.g see [Mahovsky 2005] sec 4.5.). In our setup, we instead consider the ray direction in $uv$ space to decide the traversal order, which in practice modifies the *down* and *next* from algorithm 1. While it noticeably decreases the number of traversed nodes, the benefit in terms of purely performance depends on the ray coherence, as shown in Table 4.

## 7 RESULTS

We now present visual results of our method using path tracing, with relevant metrics and comparisons. We considered the scenes shown in Figure 11. They all display high resolution displacement maps with varying tiling factors and non planar base surfaces. All renderings and timings were performed and recorded on a laptop using a RTX 2080 Max-Q GPU with 8 GB of memory and an i7-9750H CPU with 6 cores.

*Performance.* We evaluate our method along four primary axis. The first one is the amount of GPU memory used to represent and intersect the geometry. The second is the visual quality of the rendering. The third is the run-time performance, evaluated by the number of rays traced per-second. The last axis is the delay when updating the acceleration data structure upon modification of the displaced object.

We compare with uniform pre-tessellation in Table 3. We subdivided, in a preprocess step, the base mesh uniformly, before applying

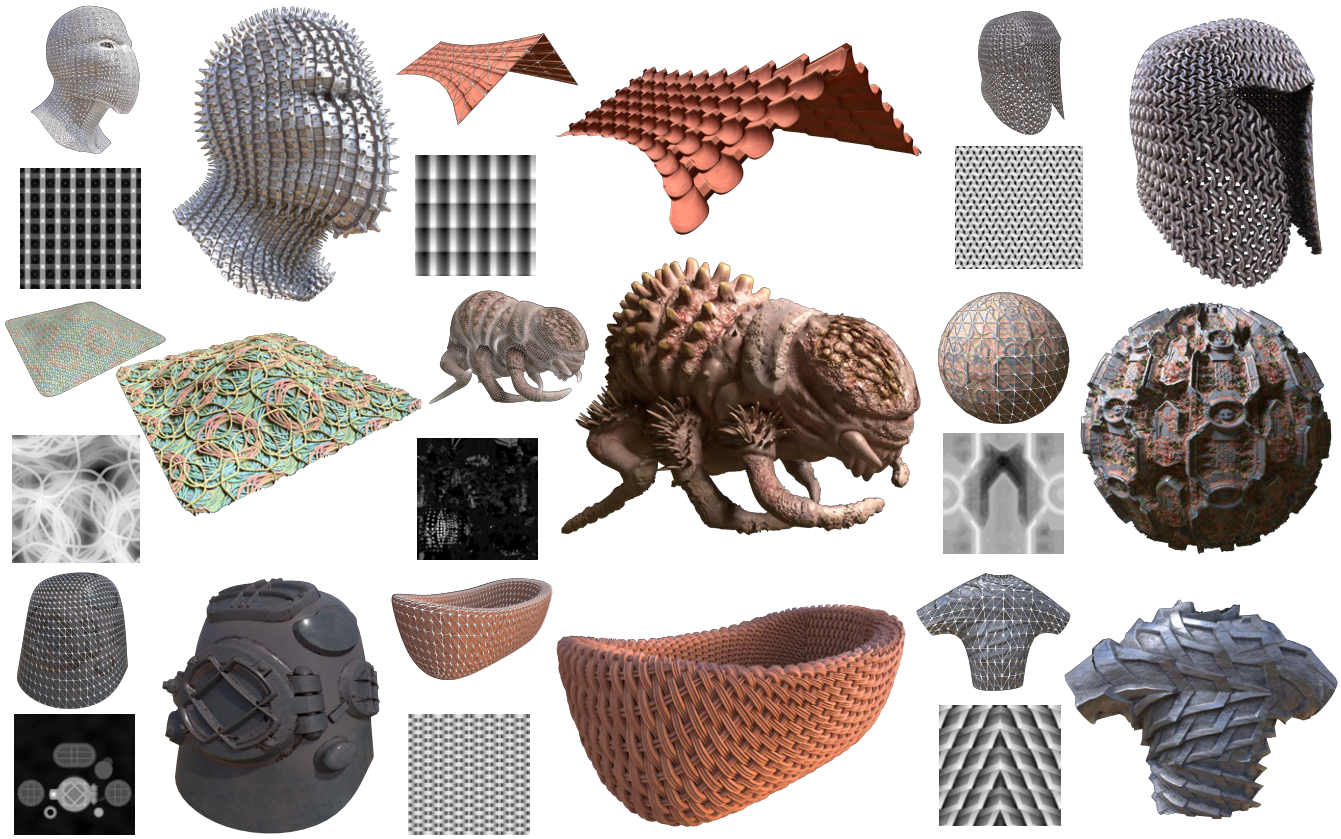

Fig. 11. **Path tracing results** illustrating various kinds of meso-structures and base surfaces. For each model, we show the base mesh (top left), the displacement map (bottom left) and the displaced surface using our method (right). From left to right, top to bottom: *Ninja* (2k texture tiled $5 \times 5$ times), *Terracotta Roof* (2k texture tiled $5 \times 5$ times), *Medieval Helmet* (2k texture tiled $5 \times 5$ times), *Fishing Ropes* (2k texture, tiled $4 \times 4$ times), *Creature* (4k texture, no tiling), *Alien Sphere* (2k texture, tiled $4 \times 8$ times), *Diving Helmet* (4k texture, no tiling), *Wicker Basket* (2k texture tiled $5 \times 5$ times), and *Elven Armor* (4k texture tile $2 \times 2$ times).

Table 3. **Comparison with uniform pre-tessellation:** Performance comparison for several test scenes between our method and alternatives, with respect to pre-process computation time, memory usage in GPU memory, update delay upon modification, and total number of rays traced per second (includes camera rays, secondary rays, and shadow rays). For the pre-tessellation, the update delay is identical to the pre-process, as any modification implies a complete pre-tellessation computation. (1) For pre-tessellation, we subdivided the base triangles uniformly until reaching the amount of GPU memory available, therefore not providing an equal-quality comparison. Pre-tessellation and our minmax mipmap preprocess were done on CPU using a single thread.

| | Tri. count | Disp. | Tiling | Uniform pre-tessellation (1) | | | Ours | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | | | | Memory | Speed | Update delay | Preprocess | Memory | Speed | Update delay |
| *Alien Sphere* | 0.9k | 2k | $4 \times 8$ | 0.9 GB | 103 Mr/s | 1.3 s | 88 ms | 34 MB | 1.8 Mr/s | 0.054 ms |
| *Wicker Basket* | 4.8k | 2k | $5 \times 5$ | 1.1GB | 126 Mr/s | 1.6 s | 91 ms | 36 MB | 1.5 Mr/s | 0.086 ms |
| *Creature* | 54k | 4k | $1 \times 1$ | 3.1 GB | 20 Mr/s | 4.6 s | 340 ms | 164 MB | 1.0 Mr/s | 0.127 ms |
| *Diving Helmet* | 2.5k | 4k | $1 \times 1$ | 2.4 GB | 124 Mr/s | 3.5 s | 397 ms | 135 MB | 2.4 Mr/s | 0.265 ms |
| *Elven Armor* | 768 | 4k | $2 \times 2$ | 0.7 GB | 133 Mr/s | 1.0 s | 353 ms | 135 MB | 1.7 Mr/s | 0.18 ms |
| *Medieval Helmet* | 2.3k | 2k | $5 \times 5$ | 2.1 GB | 104 Mr/s | 3.0 s | 80 ms | 35 MB | 2.4 Mr/s | 0.085 ms |
| *Ninja* | 8.8k | 2k | $5 \times 5$ | 2.0 GB | 25 Mr/s | 3.2 s | 83 ms | 38 MB | 0.7 Mr/s | 0.082 ms |
| *Terracotta Roof* | 8.8k | 2k | $5 \times 5$ | 0.5 GB | 30 Mr/s | 0.6 s | 80 ms | 34 MB | 4.8 Mr/s | 0.040 ms |

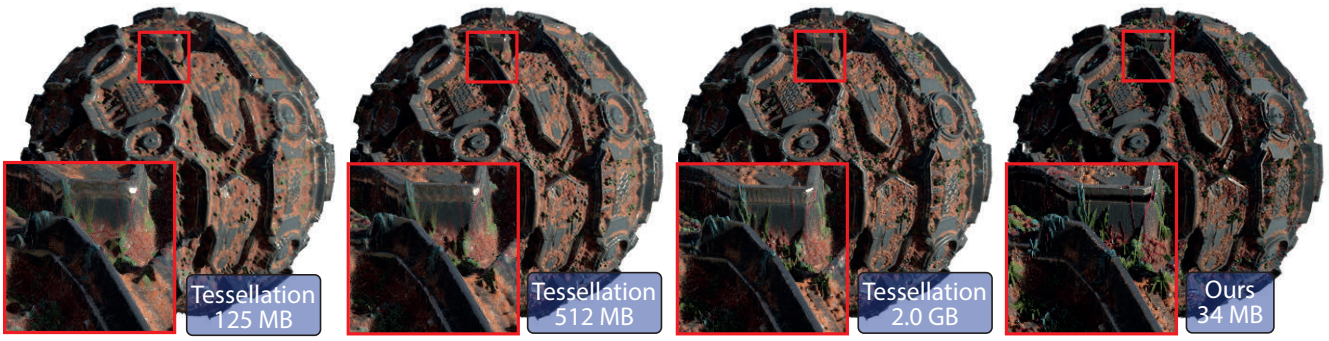


Fig. 12. **Quality comparison with uniform pre-tessellation:** Rendering results for different amounts of uniform pre-tessellation and our method, with the corresponding GPU memory used for geometry. For tessellation, the geometry consists of the index and vertex buffer. For our method, the geometry combines the base mesh index and vertex buffer, the displaced BVH, and the displaced triangle buffer.

Table 4. **Ablation study:** Relative run time performance for different versions of the traversal components with respect to the reference, higher than one meaning faster than the reference. The quantity compared is the number of rays traced by unit of time. For each scene, the first row only counts primary rays, while the second row counts primary, secondary, and shadow rays.

| | Roots (vs 4) | Texel discard (vs our custom test) | | | | Traversal order (vs 8) | | Local intersection (vs bicubic) | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | Single | None | Clipping | Sep. plane | GKJ [1988] | Fixed | Four | Leaf box | Two tri. | Bilinear |
| Alien sphere | ×0.96 | ×0.06 | ×0.71 | ×0.75 | ×0.74 | ×0.80 | ×0.99 | ×1.64 | ×1.26 | ×1.09 |
| | ×0.97 | ×0.17 | ×0.87 | ×0.90 | ×0.92 | ×0.96 | ×0.98 | ×1.57 | ×1.27 | ×1.13 |
| Creature | ×0.97 | ×0.23 | ×0.85 | ×0.81 | ×0.89 | ×1.01 | ×1.02 | ×1.56 | ×1.16 | ×1.08 |
| | ×0.93 | ×0.36 | ×0.88 | ×0.83 | ×0.93 | ×1.05 | ×0.99 | ×1.38 | ×1.10 | ×1.04 |
| Ninja | ×0.92 | ×0.24 | ×0.76 | ×0.76 | ×0.78 | ×0.96 | ×1.00 | ×1.29 | ×1.10 | ×1.05 |
| | ×0.95 | ×0.43 | ×0.87 | ×0.87 | ×0.93 | ×1.06 | ×1.00 | ×1.26 | ×1.12 | ×1.06 |
| Elven armor | ×0.96 | ×0.22 | ×0.59 | ×0.71 | ×0.68 | ×0.96 | ×1.01 | ×1.41 | ×1.15 | ×1.07 |
| | ×0.99 | ×0.42 | ×0.71 | ×0.89 | ×0.84 | ×1.07 | ×0.97 | ×1.38 | ×1.16 | ×1.09 |

the displacement per vertex. By pre-computing the displaced surface and relying on hardware-accelerated data structures, tessellation achieves very high run time performance, being two orders of magnitude faster on average than our method. However, as shown in Figure 12, the subdivision scheme consumes large amounts of memory, and cannot match the visual fidelity of our method even when using all available 8 GB of GPU memory. Moreover, since the displacement is fully baked into the base surface, any displacement modification triggers a re-tessellation, which takes typically several seconds to complete, taking in account both the tessellation process and the BVH rebuilt. In contrast, our method only needs to update the mapping between the D-BVH and the base mesh, which takes less than a 0.3 ms with a base mesh with 50k vertices.

*Ablation study.* As described in section 4, we made several design choices for our traversal. We show in Table 4 relative run time performances for different versions of the traversal. Choosing four roots instead of one gives a constant, while small, performance gain. Performing a texel discard gives a very noticeable performance boost. Its benefit varies depending on which node-triangle collision test is used. Using a traversal order based on the ray direction decreases significantly the number of traversed nodes. However, the nature of the current GPU architecture makes a divergence of traversal order between rays not always beneficial in terms of performance. Considering local intersections, the performance is indeed directly related to the complexity of the intersection test. We note, however, a relatively small difference between iterative and non-iterative intersection tests, meaning the cost of traversal has more impact on the overall performance than time spent intersecting at the leaves.

*Additional Comparisons.* We compare our method to *Relief Mapping* [Policarpo et al. 2005], which shares with our approach the negligible update delay (no heavy preprocess of the displacement map), as typically requested in our target dynamic application usages. As can be seen in Figure 13, *Relief Mapping* cannot avoid aliasing artefacts, which even degrade further the rendering quality with grazing view angle or highly non-flat base domains. On the contrary, our approach, although more computationally intensive, is free from artifact and reproduces perfectly the underlying displacement.

Moreover, as shown in Figure 14, our method naturally handles arbitrary ray-surface interaction, enabling e.g., Monte Carlo Path Tracing with local inter-reflections within the displaced geometry.

We also compared our approach to an explicit adaptive tessellation performed on a per-frame basis. We choose to exclude the actual adaptive tessellation step from our performance measure and only report timings regarding the per-frame BLAS generation requested by dynamic scenarios (see Figure 15). We use an adaptive tessellation

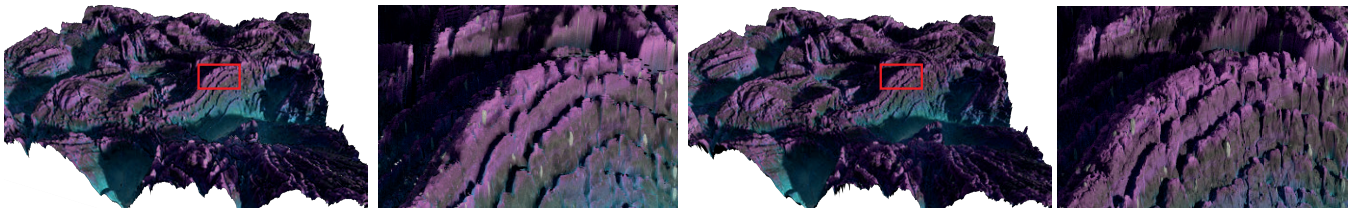

Fig. 13. **Comparison to Relief Mapping.** Although faster, relief mapping rendering reveals numerous artifacts, both in steep regions and grazing angles. On the contrary, our method (right) provide artifact-free images, suitable to assess interactively the quality and precise features of the displacement map.

Fig. 14. **Path tracing and local inter-reflections**. Our method is compatible with full Monte Carlo rendering, with all light transport effects being reproduced – here showing the *Creature* from Figure 11 using volumetric path-traced random-walk subsurface scattering (left), and a translucent glass material (right).

scheme designed to maximize quality: essentially, we first tessellate at extremely high resolution, then displace the resulting mesh before simplifying it using a state-of-the-art feature-preserving simplification based on iterative quadric-error-driven edge collapse. In this example, while our method induces a BLAS update performed in 0.082 ms prior to rendering the frame, the same update takes 35ms with the explicit adaptive tessellation. This severe degradation in hardware-supported ray tracing occurs as soon as a component is not static (base mesh deformation, displacement content, UV modifications, etc), when the BLAS cannot be factored out over time. For a fair comparison, we generated an adaptive tessellation which matches the memory footprint of our method. In spite of the (long) offline adaptive tessellation, we can easily observe in Figure 15 that numerous artefacts appear, while our method perfectly reproduces the displacement content. As future work, in the context of path tracing, the notion of explicit global adaptive tessellation could be further developed as the scene's geometry may undergo more or less aggressive decimation depending on both visibility and appearance models [Reich et al. 2015].

To some extent, it is worth noting that our method, when set up in local triangulation mode (section 5), can be understood as an implicit, per-ray adaptive tessellation scheme.

*Interactivity.* In the accompanying supplemental video, we show examples of interactive displacement authoring scenarios, including modification of the *uv* mapping, and change of displacement parameters.

Fig. 15. **Comparison to adaptive tessellation** with matching memory footprint (37MB). From left to right: base mesh, adaptive tessellation (with close-up) and our method (with close-up).

## 8 DISCUSSION

We presented a novel approach for efficiently ray-tracing displacement maps. Our method enables tiling and instancing of displacement map information that can be used across multiple base domain meshes, leading to memory savings. By avoiding pre-tessellation, we achieve much higher quality rendering with a much lower memory budget. Our approach is robust, and does not impose conditions on the displacement magnitude with respect to curvature of the base domain. We note, however, a few limitations with our current approach and a number of future directions.

*Seams.* Our current system does not guarantee a watertight solution and cracks may appear along *uv* chart boundaries. The generation of seam free *uv* maps is an orthogonal issue to our method and not unique to our approach. Some solutions exist, for example a pre-process can be applied to the height field to eliminate seams [Liu et al. 2017]. Unfortunately this approach does not work with tiling, or instanced use of the displacement map across multiple meshes. It is also too slow for interactive displacement map design. Supporting computationally efficient seam free displacement mapping in a more general context remains an open and challenging problem requiring more exploration.

*Fixed Function Hardware Support.* Modern GPUs support custom acceleration for hierarchical bounding volume traversal and ray-triangle intersection using specific hardware cores [NVIDIA 2020]. Our current method is unable to take full advantage of this hardware as the D-BVH traversal uses an intersection shader which operates on generic compute shaders. This partially explains the large performance gap we see relative to pre-tessellation methods. A future extension would be to extend modern hardware to directly support displacement mapping. We also think further enhancements can be made such as computing compressed but conservative min/max hierarchies or applying per-texel affine arithmetic instead of intervals for tighter bounds, bringing even better performance.

*Memory Streaming.* Our approach can directly render from displacement information stored in GPU texture mip-maps. Given that our method supports level-of-detail by stopping at higher levels of the mip-map, the entire pyramid of displacement information does not need to be memory resident. This opens up the ability to stream geometric detail during ray-tracing by leveraging *sparse texture* functionality supported in modern GPU hardware [Burgess 2020].

*Extensions.* We see a number of potential explorations for our method. We believe our approach can be extended to support vector displacement, procedural displacement maps, volumetric materials, or more complex base surfaces. This property comes from our method's ability to dynamically construct a general acceleration hierarchy that can bound arbitrary intersect-able content. It would be interesting to explore recursive nesting of our structure to achieve infinite fractal scale detail.

## 9 CONCLUSION

Our algorithm trades compute for memory. We believe that this aligns well with modern hardware advances where bandwidth relative to compute is growing more slowly. Our approach directly operates on the displacement map memory without expensive preprocessing, making displacement maps as natural to create, modify and render as the other maps modeling modern digital materials. Efficiently managing geometric complexity in the context of ray-tracing continues to be challenging, however, we believe our approach opens up new opportunities for increased visual fidelity as ray tracing goes real-time.

## ACKNOWLEDGMENTS

We thank Luc Chamerlat for helping with the supplemental and providing the *Creature* asset and the teaser scene; Peter Kutz for his significant contribution to the code needed to render Figure 14; and Krishna Mullia for helping with experiments.

## REFERENCES

Lionel Baboud, Elmar Eisemann, and Hans-Peter Seidel. 2011. Precomputed safety shapes for efficient and accurate height-field rendering. *IEEE transactions on visualization and computer graphics* 18, 11 (2011), 1811–1823.

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*. 5–12.

James F Blinn. 1978. Simulation of wrinkled surfaces. *ACM SIGGRAPH computer graphics* 12, 3 (1978), 286–292.

J. Burgess. 2020. RTX on—The NVIDIA Turing GPU. *IEEE Micro* 40, 2 (2020), 36–44. https://doi.org/10.1109/MM.2020.2971677

Nathan A Carr, Jared Hoberock, Keenan Crane, and John C Hart. 2006. Fast GPU ray tracing of dynamic meshes using geometry images.. In *Graphics Interface*, Vol. 2006. Citeseer, 203–209.

Ying-Chieh Chen and Chun-Fa Chang. 2008. A Prism-Free Method for Silhouette Rendering in Inverse Displacement Mapping. *Comput. Graph. Forum* 27 (10 2008), 1929–1936. https://doi.org/10.1111/j.1467-8659.2008.01341.x

Per H. Christensen, David M. Laur, Julia Fong, Wayne L. Wooten, and Dana Batali. 2003. Ray Differentials and Multiresolution Geometry Caching for Distribution Ray Tracing in Complex Scenes. *Computer Graphics Forum* 22, 3 (2003), 543–552. https://doi.org/10.1111/1467-8659.t01-1-00702 arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1111/1467-8659.t01-1-00702

Robert L Cook. 1984. Shade trees. In *Proceedings of the 11th annual conference on Computer graphics and interactive techniques*. 223–231.

C. Dachsbacher and Natalya Tatarchuk. 2007. Prism Parallax Occlusion Mapping with Accurate Silhouette Generation.

Luiz Henrique De Figueiredo and Jorge Stolfi. 2004. Affine arithmetic: concepts and applications. *Numerical Algorithms* 37, 1-4 (2004), 147–158.

Peter Djeu, Warren Hunt, Rui Wang, Ikrima Elhassan, Gordon Stoll, and William R. Mark. 2011. Razor: An Architecture for Dynamic Multiresolution Ray Tracing. *ACM Trans. Graph.* 30, 5, Article 115 (Oct. 2011), 26 pages. https://doi.org/10.1145/2019627.2019634

William Donnelly. 2005. Per-pixel displacement mapping with distance functions. *GPU gems* 2, 22 (2005), 3.

Tom Duff, James Burgess, Per Christensen, Christophe Hery, Andrew Kensler, Max Liani, and Ryusuke Villemin. 2017. Building an orthonormal basis, revisited. *Journal of Computer Graphics Techniques (JCGT)* 6, 1 (2017).

Jonathan Dummer. 2006. Cone step mapping: An iterative ray-heightfield intersection algorithm. *URL: http://www. lonesock. net/files/ConeStepMapping. pdf* 2, 3 (2006), 4.

Elmer G Gilbert, Daniel W Johnson, and S Sathiya Keerthi. 1988. A fast procedure for computing the distance between complex objects in three-dimensional space. *IEEE Journal on Robotics and Automation* 4, 2 (1988), 193–203.

Johannes Hanika, Alexander Keller, and Hendrik P. A. Lensch. 2010. Two-Level Ray Tracing with Reordering for Highly Complex Scenes. In *Proceedings of Graphics Interface 2010* (Ottawa, Ontario, Canada) *(GI '10)*. Canadian Information Processing Society, CAN, 145–152.

John C Hart. 1996. Sphere tracing: A geometric method for the antialiased ray tracing of implicit surfaces. *The Visual Computer* 12, 10 (1996), 527–545.

W. Heidrich and H. Seidel. 1998. Ray-tracing Procedural Displacement Shaders. In *Graphics Interface*.

Johannes Hirche, Alexander Ehlert, Stefan Guthe, and Michael Doggett. 2004. Hardware Accelerated Per-Pixel Displacement Mapping. In *Proceedings of Graphics Interface 2004* (London, Ontario, Canada) *(GI '04)*. Canadian Human-Computer Communications Society, Waterloo, CAN, 153–158.

Qiming Hou, Hao Qin, Wenyao Li, Baining Guo, and Kun Zhou. 2010. Micropolygon Ray Tracing with Defocus and Motion Blur. In *ACM SIGGRAPH 2010 Papers* (Los Angeles, California) *(SIGGRAPH '10)*. Association for Computing Machinery, New York, NY, USA, Article 64, 10 pages. https://doi.org/10.1145/1833349.1778801

Warren Hunt, William R. Mark, and Don Fussell. 2007. Fast and Lazy Build of Acceleration Structures from Scene Hierarchies. In *2007 IEEE Symposium on Interactive Ray Tracing*. 47–54. https://doi.org/10.1109/RT.2007.4342590

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.

Tomomichi Kaneko, Toshiyuki Takahei, Masahiko Inami, Naoki Kawakami, Yasuyuki Yanagida, Taro Maeda, and Susumu Tachi. 2001. Detailed shape representation with parallax mapping. In *Proceedings of ICAT*, Vol. 2001. 205–208.

Khronos. 2020. *Vulkan Ray Tracing specification.* https://www.khronos.org/blog/vulkan-ray-tracing-final-specification-release

L. Lee, Shih-Wei Tseng, and W. Tai. 2009. Improved Relief Texture Mapping Using Minmax Texture. *2009 Fifth International Conference on Image and Graphics* (2009), 547–552.

Alexander Lier, Magdalena Martinek, Marc Stamminger, and Kai Selgrad. 2018. A High-Resolution Compression Scheme for Ray Tracing Subdivision Surfaces with Displacement. *Proceedings of the ACM on Computer Graphics and Interactive Techniques* 1, 2 (2018), 1–17.

Songrun Liu, Zachary Ferguson, Alec Jacobson, and Yotam Gingold. 2017. Seamless: Seam erasure and seam-aware decoupling of shape from mesh resolution. *ACM Transactions on Graphics (TOG)* 36, 6, Article 216 (Nov. 2017), 15 pages. https://doi.org/10.1145/3130800.3130897

The-Kiet Lu, K. Low, and J. Zheng. 2009. Fast visualization of complex 3D models using displacement mapping. In *Graphics Interface*.

Jeffrey A. Mahovsky. 2005. *Ray Tracing with Reduced-Precision Bounding Volume Hierarchies.* Ph.D. Dissertation. CAN.

William Martin, Elaine Cohen, Russell Fish, and Peter Shirley. 2000. Practical ray tracing of trimmed NURBS surfaces. *Journal of Graphics Tools* 5, 1 (2000), 27–52.

Tomas Möller and Ben Trumbore. 1997. Fast, minimum storage ray-triangle intersection. *Journal of graphics tools* 2, 1 (1997), 21–28.

Henry Moreton. 2001. Watertight Tessellation Using Forward Differencing. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Workshop on Graphics Hardware* (Los Angeles, California, USA) *(HWWS '01)*. Association for Computing Machinery, New York, NY, USA, 25–32. https://doi.org/10.1145/383507.383520

K. Moule and M. McCool. 2002. Efficient Bounded Adaptive Tessellation of Displacement Maps. In *Graphics Interface*.

J. Munkberg, J. Hasselgren, Robert Toth, and T. Akenine-Möller. 2010. Efficient bounding of displaced Bézier patches. In *HPG '10*.

M. Nießner and C. Loop. 2013. Analytic Displacement Mapping using Hardware Tessellation. *ACM Transactions on Graphics (TOG)* 32, 3 (2013), 26.

Matthias Nießner, Charles Loop, Mark Meyer, and Tony Derose. 2012. Feature-Adaptive GPU Rendering of Catmull-Clark Subdivision Surfaces. *ACM Trans. Graph.* 31, 1, Article 6 (Feb. 2012), 11 pages. https://doi.org/10.1145/2077341.2077347

NVIDIA. 2020. *RTX Technology - RT Cores.* https://developer.nvidia.com/rtx/raytracing#rtcores

Kyoungsu Oh, Hyunwoo Ki, and Cheol-Hi Lee. 2006. Pyramidal displacement mapping: a gpu based artifacts-free ray tracing through an image pyramid. In *Proceedings of the ACM symposium on Virtual reality software and technology*. 75–82.

Manuel Oliveira and Fabio Policarpo. 2005. An Efficient Representation for Surface Details. *UFRGS Technical Report RP-351* (01 2005).

Afonso Paiva, Filipe de Carvalho Nascimento, Luiz Henrique de Figueiredo, and Jorge Stolfi. 2012. Approximating implicit curves on triangulations with affine arithmetic. In *2012 25th SIBGRAPI Conference on Graphics, Patterns and Images*. IEEE, 94–101.

Matt Pharr and Pat Hanrahan. 1996. Geometry Caching for Ray-Tracing Displacement Maps. In *Proceedings of the Eurographics Workshop on Rendering Techniques '96* (Porto, Portugal). Springer-Verlag, Berlin, Heidelberg, 31–ff.

M. Pharr, Craig E. Kolb, Reid Gershbein, and P. Hanrahan. 1997. Rendering complex scenes with memory-coherent ray tracing. *Proceedings of the 24th annual conference on Computer graphics and interactive techniques* (1997).

Fabio Policarpo and Manuel M Oliveira. 2007. Relaxed cone stepping for relief mapping. *GPU gems* 3 (2007), 409–428.

Fábio Policarpo, Manuel M Oliveira, and João LD Comba. 2005. Real-time relief mapping on arbitrary polygonal surfaces. In *Proceedings of the 2005 symposium on Interactive 3D graphics and games*. 155–162.

Serban D. Porumbescu, Brian Budge, Louis Feng, and Kenneth I. Joy. 2005. Shell Maps. *ACM Trans. Graph.* 24, 3 (July 2005), 626–633. https://doi.org/10.1145/1073204.1073239

Andreas Reich, Tobias Günther, and Thorsten Grosch. 2015. Illumination-driven Mesh Reduction for Accelerating Light Transport Simulations. *Computer Graphics Forum* 34, 4 (2015), 165–174.

Siegfried M Rump and Masahide Kashiwagi. 2015. Implementation and improvements of affine arithmetic. *Nonlinear Theory and Its Applications, IEICE* 6, 3 (2015), 341–359.

Kai Selgrad, Alexander Lier, Magdalena Martinek, Christoph Buchenau, Michael Guthe, Franziska Kranz, Henry Schäfer, and Marc Stamminger. 2016. A Compressed Representation for Ray Tracing Parametric Surfaces. *ACM Trans. Graph.* 36, 1, Article 5 (2016), 13 pages.

Christian Sigg and Markus Hadwiger. 2005. Fast third-order texture filtering. *GPU gems* 2 (2005), 313–329.

Brian Smits, Peter Shirley, and Michael M Stark. 2000. Direct ray tracing of displacement mapped triangles. In *Eurographics Workshop on Rendering Techniques*. Springer, 307–318.

L. Szirmay-Kalos and Tamás Umenhoffer. 2008. Displacement Mapping on the GPU — State of the Art. *Computer Graphics Forum* 27 (2008).

László Szirmay-Kalos, Tamás Umenhoffer, Gustavo Patow, László Szécsi, and Mateu Sbert. 2009. Specular effects on the gpu: State of the art. In *Computer Graphics Forum*, Vol. 28. Wiley Online Library, 1586–1617.

Natalya Tatarchuk. 2006. Dynamic parallax occlusion mapping with approximate soft shadows. In *Proceedings of the 2006 symposium on Interactive 3D graphics and games*. 63–69.

Art Tevs, Ivo Ihrke, and Hans-Peter Seidel. 2008. Maximum mipmaps for fast, accurate, and scalable dynamic height field rendering. In *Proceedings of the 2008 symposium on Interactive 3D graphics and games*. 183–190.

Ingo Wald, Sven Woop, Carsten Benthin, Gregory S Johnson, and Manfred Ernst. 2014. Embree: a kernel framework for efficient CPU ray tracing. *ACM Transactions on Graphics (TOG)* 33, 4 (2014), 1–8.

Lifeng Wang, Xi Wang, Xin Tong, Stephen Lin, Shimin Hu, Baining Guo, and Heung-Yeung Shum. 2003. View-dependent displacement mapping. *ACM Transactions on graphics (TOG)* 22, 3 (2003), 334–339.

X. Wang, J. Maillot, E. Fiume, V. Ng-Thow-Hing, Andrew Woo, and S. Bakshi. 2000. Feature-based Displacement Mapping. In *Rendering Techniques*.

Xi Wang, Xin Tong, Stephen Lin, Shimin Hu, Baining Guo, and Heung-Yeung Shum. 2004. Generalized displacement maps. In *Proceedings of the Fifteenth Eurographics conference on Rendering Techniques*. 227–233.

Amy Williams, Steve Barrus, R Keith Morley, and Peter Shirley. 2005. An efficient and robust ray-box intersection algorithm. In *ACM SIGGRAPH 2005 Courses*. 9–es.

Keith Yerex and Martin Jägersand. 2004. Displacement mapping with ray-casting in hardware.. In *SIGGRAPH sketches*. 149.

## A AFFINE ARITHMETIC

Affine arithmetic represents quantities as affine combinations of symbolic variables whose values lie in the range $[-1, 1]$ and which describe either sources of variation in the data or approximations made during computations. In our case, for simplicity and efficiency, we only keep track of the coefficients from two symbolic variables $\epsilon_u$ and $\epsilon_v$, related to the texture coordinates $u$ and $v$. All approximations not related to the two previous symbolic variables encountered during computations are merged into a single symbolic variable $\epsilon_K$. Therefore, any scalar or vector quantity $x$ considered can be written as:

$$[x] = [x_c + x_u\epsilon_u + x_v\epsilon_v + x_K\epsilon_K] \tag{10}$$

where $x_c, x_u, x_v, x_K$ are constants with the same number of channels as $x$. For readability purposes, we will omit the symbolic variables for the rest of the section and write an affine form $x$ as $[x_c, x_u, x_v, x_K]$.

The affine form associated to the height relative to a texel can be retrieved from a single minmax mipmap fetch:

$$[h] = \frac{1}{2}\left[\overline{M}^k_{i,j} + \underline{M}^k_{i,j}, 0, 0, \overline{M}^k_{i,j} - \underline{M}^k_{i,j}\right] \tag{11}$$

Note that since we only retrieve an interval for the height instead of a complete affine form, any possible correlation between the height values and the base surface is lost.

As our $uv$ domain $\Omega^k_{i,j}$ is simply a 2D axis-aligned box, the $uv$ affine form can be easily computed as:

$$[uv] = 2^k\left[\left(\frac{j+1/2}{W}, \frac{i+1/2}{H}\right), \left(\frac{1}{2W}, 0\right), \left(0, \frac{1}{2H}\right), (0,0)\right] \tag{12}$$

Combining in Equation 1 the affine forms for displacement, base interpolated position and unit normal, we obtain an affine form for the displaced surface relatively to a given texel $[\mathbf{S}]$, from which we recover an AABB for the displaced surface:

$$\mathbf{S}(u,v) \in [\mathbf{S}_c - |\mathbf{S}_u| - |\mathbf{S}_v| - |\mathbf{S}_K|, \mathbf{S}_c + |\mathbf{S}_u| + |\mathbf{S}_v| + |\mathbf{S}_K|] \tag{13}$$

where $|\cdot|$ is the per-channel absolute value.

We now detail formulas used for affine arithmetic computations in the paper. The affine form of a constant quantity is simply $[x_c, 0, 0, 0]$. As the $\epsilon_K$ term combines all approximations made during computations, the amplitude of $x_K$ can only increase after an operation. Therefore we chose to keep $x_K \geq 0$ (component-wise) in all formulas. Addition is applied component-wise and $\cdot$ denotes the dot product. Addition between two affine forms:

$$\begin{bmatrix} x_c \\ x_u \\ x_v \\ x_K \end{bmatrix} + \begin{bmatrix} y_c \\ y_u \\ y_v \\ y_K \end{bmatrix} = \begin{bmatrix} x_c + y_c \\ x_u + y_u \\ x_v + y_v \\ x_K + y_K \end{bmatrix}$$

Dot product between two vector affine forms:

$$\begin{bmatrix} x_c \\ x_u \\ x_v \\ x_K \end{bmatrix} \cdot \begin{bmatrix} y_c \\ y_u \\ y_v \\ y_K \end{bmatrix} = \begin{bmatrix} x_c \cdot y_c \\ x_u \cdot y_c + x_c \cdot y_u \\ x_v \cdot y_c + x_c \cdot y_v \\ |x_K \cdot y_c| + |x_c \cdot y_K| + \\ (|x_u| + |x_v| + x_K) \cdot (|y_u| + |y_v| + y_K) \end{bmatrix}$$

Scalar multiplication and matrix-vector product can be directly obtained from the dot product formula.

Squared norm:

$$\begin{bmatrix} x_c \\ x_u \\ x_v \\ x_K \end{bmatrix} \cdot \begin{bmatrix} x_c \\ x_u \\ x_v \\ x_K \end{bmatrix} = \begin{bmatrix} \|x_c\|^2 + \frac{1}{2}\||x_u| + |x_v| + x_K\|^2 \\ 2x_c \cdot x_u \\ 2x_c \cdot x_v \\ 2|x_c \cdot x_K| + \frac{1}{2}\||x_u| + |x_v| + x_K\|^2 \end{bmatrix}$$

Note that the $\epsilon_K$ term is smaller using the squared norm formula than using the dot product of a form with itself.

To apply a general non-linear unary operator $f$ to an affine form $[x]$, a linear approximation of $f$ on the domain of $x$ must be derived. More specifically, one must find constant values $\alpha$, $\gamma$, and $\delta$ such that $|f(t) - (\alpha \cdot t + \gamma)| \leq \delta$ for all $t \in [\underline{x}, \overline{x}]$. An affine form for $f([x])$ can then be obtained using:

$$f([x]) = \begin{bmatrix} \alpha \cdot x_c + \gamma \\ \alpha \cdot x_u \\ \alpha \cdot x_v \\ |\alpha \cdot x_K| + \delta \end{bmatrix} \tag{14}$$

There are many possibilities for affine approximations, but two choices are of particular interest. As detailed in De Figueiredo and Stolfi [2004], the *Min-Range* approximation minimizes the approximation bounding volume while the *Chebyshev* one minimizes $\delta$. In case the function is concave or convex, closed-form formulas for both approximations can be derived [Rump and Kashiwagi 2015].

We approximate $f(x) = \text{clamp}(x, \cdot, \cdot)$ using its *Chebyshev* approximation, and $g(x) = \frac{1}{\sqrt{x}}$ using its *Min-Range* approximation:

| | $f$ | $g$ |
|---|---|---|
| $\alpha$ | $\frac{f(\overline{x})-f(\underline{x})}{\overline{x}-\underline{x}}$ | $-\frac{1}{2}g(\overline{x})^3$ |
| $\gamma$ | $\frac{1}{2}(1-\alpha)\left(f(\overline{x})+f(\underline{x})\right)$ | $\frac{1}{2}\left(g(\underline{x})+g(\overline{x})-\alpha\left(\underline{x}+\overline{x}\right)\right)$ |
| $\delta$ | $(1-\alpha)f(\overline{x})-\gamma$ | $\frac{1}{2}\left\|g(\underline{x})-g(\overline{x})-\alpha\left(\underline{x}-\overline{x}\right)\right\|$ |

(15)

## B ITERATIVE INTERSECTION TESTS

In this appendix only, [ ] simply denotes matrices or column vectors and not affine forms or bounding boxes. For a given base triangle, the surface defined in Equation 1 is a parametric surface that can be intersected using iterative methods. Following the work of Martin et al. [2000], we consider the following equations for a query ray with direction $d$ and origin $o$:

$$F(u,v) = \begin{bmatrix} (S(u,v)-o)\cdot d_1 \\ (S(u,v)-o)\cdot d_2 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix} \tag{16}$$

where $d, d_1, d_2$ forms an orthonormal basis, obtained for example using the routine of Duff et al. [2017]. Starting from $(u_0, v_0)$ at the leaf center, we solve the previous equation using Newton's iterative method, $F$ having simple derivatives with respect to $u$ and $v$:

$$\begin{aligned} J_F &= \left[\frac{\partial F}{\partial u}, \frac{\partial F}{\partial u}\right] = \begin{bmatrix} \frac{\partial S}{\partial u}\cdot d_1, \frac{\partial S}{\partial v}\cdot d_1 \\ \frac{\partial S}{\partial u}\cdot d_2, \frac{\partial S}{\partial v}\cdot d_2 \end{bmatrix} \\ J_S &= \left[\frac{\partial S}{\partial u}, \frac{\partial S}{\partial u}\right] = J_P + \hat{N}J_h + hJ_{\hat{N}} \\ &= J_P + \hat{N}J_h + \frac{h}{\|N\|}\left(J_N - \hat{N}\left[\frac{\partial N}{\partial u}\cdot\hat{N}, \frac{\partial N}{\partial v}\cdot\hat{N}\right]\right) \end{aligned} \tag{17}$$

where $J$ denotes the Jacobian matrix with respect to $u$ and $v$. In case the base surface is purely a triangle mesh, $J_P$ and $J_N$ are $3 \times 2$ matrices constant per base triangle as seen from Equation 7. $J_h$ can be computed explicitly using texel values in a neighborhood of the $uv$ point of interest, for both bilinear and bicubic sampling [Sigg and Hadwiger 2005].

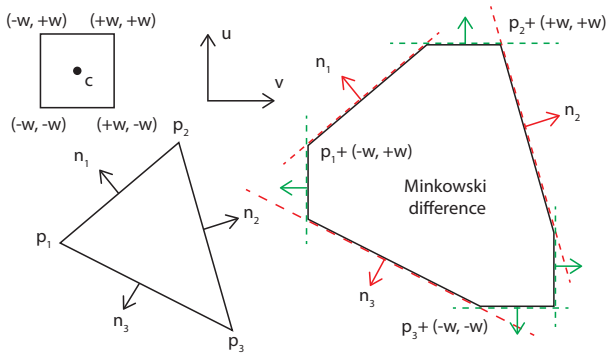


Fig. 16. **Triangle-square 2D configuration.** The Minkowski difference between a square and a triangle in 2D is made of at most seven edges. Testing for a collision means testing on which half plane the origin is for each of those edges. Four of them are equivalent to check whether the square and the triangle bounding box intersect (green half-spaces). The remaining three are equivalent to check if any triangle edge is a separating axis, which can be done by considering for each triangle edge a carefully chosen box corner (red half-spaces).

**Algorithm 4:** Triangle-square 2D collision.
The triangle is defined by its vertices $p_1, p_2, p_3$ and its outward-pointing edges normals $n_1, n_2, n_3$. The square is defined by its center $c$ and its side half-length $w$. In practice we pre-compute $p_i$, $n_i$, and $\text{minmax}_i\, p_i$ at the beginning of the intersection shader, as the collision function is always called with the same triangle. Ternary, comparison, and min-max operators are applied component-wise, and `contains` is a point-in-triangle routine in 2D.

**Function** `collision`$(p_i, n_i, c, w)$**:**
  $p_i \mathrel{-}= c$ `// As if the square is origin-centered`
  **if any** $\min((w,w), \max_i p_i) \leq \max(-(w,w), \min_i p_i)$
   **or any** `dot` $(n_i, p_i + (n_i \geq 0\ ?\ w : -w)) \leq 0$ **then**
    **return** SquareOutsideTriangle
  **if** $(p_1, p_2, p_3)$ `contains` **all** $(\pm w, \pm w)$ **then**
    **return** SquareInsideTriangle
  **return** SquareOverlappingTriangle

## C TRIANGLE-SQUARE COLLISION

The general version of the GJK algorithm [Gilbert et al. 1988] computes collision between two convex shapes by determining if the origin is contained in their Minkowski difference in an iterative fashion. But in our scenario with a triangle and a square in 2D, the Minkowski difference is made of at most seven edges, and all these origin-edge tests can be written explicitly as seen in Figure 16. The final routine is described in algorithm 4.