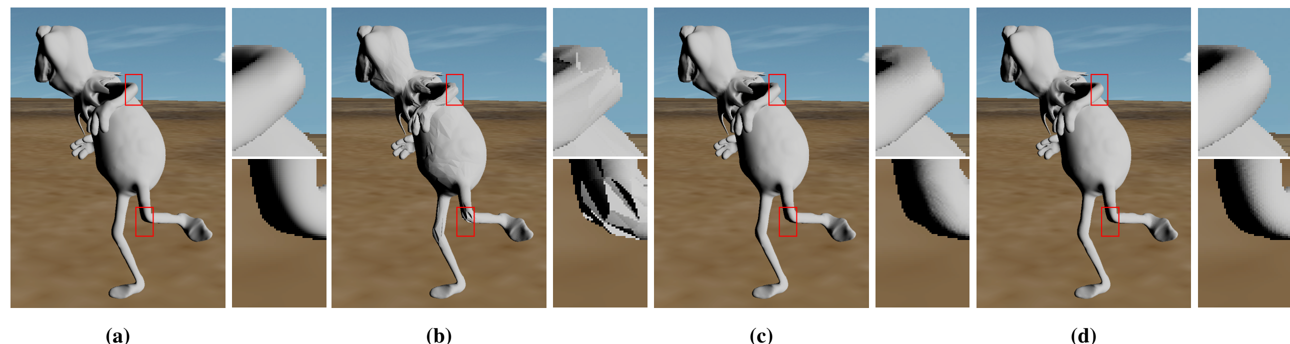
Pacific Graphics 2024
Author Version

COMPUTER GRAPHICS *forum*
Volume 0 (1981), Number 0

Ray Tracing Animated Displaced Micro-Meshes

Holger Gruen Carsten Benthin Andrew Kensler Joshua Barczak David McAllister

Advanced Micro Devices, Inc.



(a) **(b)** **(c)** **(d)**

Figure 1: *The high resolution model (memory size ∼10 MB) has been simplified to a DMM base mesh that reduces triangle count by a factor of 64, which amounts to a DMM of subdivision level 3. The combined memory size of the base mesh and the displacements amounts to ∼0.5 MB. (a) Original high resolution frog model (∼400k triangles) animated through* skinning *with magnified elbow and knee zoom-ins. (b) Standard animated DMMs (∼6k base triangles), micro-triangle normals used for flat shading. (c) Standard animated DMMs (∼6k base triangles), using our interpolated matrix approach to compute micro-triangle normals (flat shaded). (d) Our interpolated matrix approach, used for geometry and micro-triangle normals (flat shaded) shows improved silhouettes and no normal artefacts.*

Abstract

We present a new method that allows efficient ray tracing of virtually artefact-free animated displaced micro-meshes (DMMs) [MMT23] and preserves their low memory footprint and low BVH build and update cost. DMMs allow for compact representation of micro-triangle geometry through hierarchical encoding of displacements. Displacements are computed with respect to a coarse base mesh and are used to displace new vertices introduced during 1 : 4 *subdivision of the base mesh. Applying non-rigid transformation to the base mesh can result in silhouette and normal artefacts (see Figure 1) during animation. We propose an approach which prevents these artefacts by interpolating transformation matrices before applying them to the DMM representation. Our interpolation-based algorithm does not change DMM data structures and it allows for efficient bounding of animated micro-triangle geometry which is essential for fast tessellation-free ray tracing of animated DMMs.*

1. Introduction

Real-time rendering has always been striving for greater geometric complexity to increase scene fidelity. Recently, displaced micro-meshes (DMMs) [MMT23] have been proposed as an efficient representation for lossily compressed geometry. The representation requires topology restrictions but achieves very high compression rates of as little as *four* bits per triangle. The DMM format is based on hierarchically encoded displacements per base triangle in combination with error correction at each subdivision level. At each subdivision level, vertices are displaced along the interpolated vertex normals of the base triangle.

Ray tracing applications use bounding volume hierarchies (BVH), see e.g., [MOB*21], to enable fast ray versus scene intersection tests. Real-time ray tracing applications, like computer games, still face two problems related to BVHs.

The first problem is that BVHs consume large amounts of memory. The second is that the time to construct a BVH from highly detailed objects can be prohibitively high. This is especially true if the detailed geometry is animated and its BVH needs to be updated or recomputed every frame. DMMs promise to alleviate both problems. Memory consumption is down because DMM geometry is highly compressed. Furthermore, the displacement data for an object comprised of DMMs need to be stored only once and can be reused by each instance of this object, even if the DMM base mesh is animated. So each new instance only adds an amount of memory that is proportional to the number of base mesh triangles. BVH build and update costs are also down as the complexity of the BVH is a function of the triangle count of the coarse DMM base mesh.

© 2024 Eurographics - The European Association
for Computer Graphics and John Wiley & Sons Ltd.

Animated DMMs can show various artefacts though. One of our main contributions is to redefine DMMs under animation to remove artefacts while preserving their low BVH memory footprints and low BVH construction costs. The other contribution is a tessellation-free algorithm for ray tracing these animated DMMs. Our approach does not change the data structures that represent DMMs and that allows for high compression rate.

The most frequently used type of animated geometry in computer games are game characters. Typically, these characters are animated using a form of skinning (see e.g., [KCvO07] and [Gru24]). Skinning techniques such as linear blend skinning (LBS) [Gru24] assign a set of transformation matrices and associated weights to each vertex that is influenced by this given set of transformations. As the transformation matrices, usually called bones in the context of LBS, are animated, updated vertex positions and normals are animated as well. In the context of this work we concentrate on skinning/animation techniques that ultimately produce a 4×4 matrix which is then used to transform or skin a vertex. LBS is probably the most popular skinning technique used in computer games and real-time applications. Note that LBS has also been used successfully to animate tree geometry and other geometry that represents plants [Kiy24].

DMM base meshes are created in a simplification scheme [MMT23] tailored to the generation of high-quality base meshes which are optimized for tessellation and displacement sampling. The result of applying this scheme is a simplified or coarse DMM base mesh with a low number of triangles, vertex positions and normals and the data that describes the DMM displacements.

The highly detailed mesh, that is the input to the DMM simplification process, needs a set of bones and weights attached to each of its vertices for it to be animatable with LBS. These matrices and weights are typically assigned and tuned by a human artist who is responsible for animating a game character. Maggiordomo et al. [MMT23] use tweaked edge collapses (see [Hop23]) during simplification. Each edge collapse eliminates two triangles and reduces the two vertices of an edge to one new vertex. The position of a new vertex is chosen to minimize some error function. As a result it is unclear what set of bones and weights to use for each new vertex.

One way to deal with this problem is to let a human artist regenerate bone and weight assignments for the simplified base mesh. DeCoro and Rusinkiewicz [DR05] extend the commonly-used quadric error metric [GH97] to simplify a mesh by incorporating knowledge of potential poses. They also present a method to update bone weights during simplification. This however can lead to a higher number of bones influencing vertices created during simplification. This is not desirable for most game applications.

Another way to deal with the issue is to only use half-edge collapses (see [LWL*12]) during simplification. Half-edge collapses ensure that one of the two vertices that participate in an edge collapse is dropped into the other destination vertex. This destination vertex does not change position and can thus keep its original bone and weights assignments. The animated base meshes of the test objects in our work uses this method to get around the problem of assigning new bones and weights during or after simplification. Half-edge collapses are an efficient way of computing lower detail approximations of objects that do not need additional artists intervention for animation and are popular among game developers.

In the context of e.g., LBS animations, the DMM representation causes issues in our test objects. The encoded displacements are computed along the interpolated normals with respect to a fixed pose (for example: the rigging pose) of the base mesh. After animating base mesh positions and normals by LBS, displacements along interpolated animated normals do not always generate an artefact-free displaced mesh (see Figure 1b).

Some of the artefacts result from the fact that the combined DMM displacements that produce a smooth looking composite surface do not produce a smooth looking composite surface after animation. If some part of the surface bends the pre-computed displacements do not change magnitude to produce the expected smooth overall surface with the expected higher curvature as shown in Figure 2. The sharp bends produced by using interpolated normals can create unwanted silhouette edges. These tend to be where the surface deforms non-rigidly because of skinning.

As displacements are pre-computed and thus static, they also do not change amplitude or signs. This means that e.g., the undulating animation of a snake which locally changes the surface from curving outwards to curving inwards can't be represented by the static

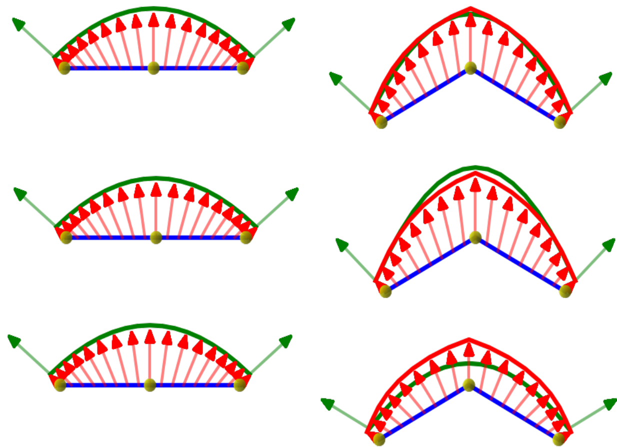


Figure 2: *(Left) 2D depiction of two edges of a DMM base mesh in the rigging pose in blue. The red arrows show interpolated base mesh normals that sample the green high resolution mesh. The length of the normals indicates the magnitude of the displacement that scale the interpolated normal. (Right) After animation, the skinned high resolution mesh (green) has bent at and around the central vertex (yellow). Scaling the interpolated skinned normals (long green arrows) of the animated base by the displacements computed for the rigging pose shows a reconstructed red surface that has a sharp bend in the middle and also each part of the composite surface lacks sufficient curvature. Depending on the animated base mesh normals, the micro-triangle surface that is reconstructed can be a close match to high resolution skinned surface, but it can also be below or above the high resolution skinned surface.*

© 2024 Eurographics - The European Association
for Computer Graphics and John Wiley & Sons Ltd.

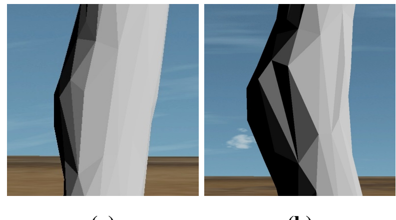


Figure 3: *(a) A close-up of the left knee of the frog model's DMM base mesh (created using half-edge collapses) in the rigging pose. (b) A close-up of the left knee of the frog model's DMM base mesh (created using half-edge collapses) skinned during animation. It already shows internal silhouette artefacts.*

displacements of standard DMMs. The general problem is that interpolated normals do not follow any real underlying surface shape as in the case of displaced subdivision surfaces [LMH00].

Some surface artefacts are generated by skinning base meshes that were created using half-edge collapses only (see Figure 3). A human artist who adjusts bone and weights assignments after simplification can probably avoid some of those. In this work we will continue to use base meshes produced by half-edge collapses, as we will show that our proposed method can even remove artefacts created by these base meshes.

On top of strong silhouette artefacts, the animated normals and tangent frames of resulting micro-triangles can generate shading artefacts. These silhouette and normal artefacts can prevent DMMs from being adopted as a compact representation for animated assets. As a result, none of the BVH build time savings and memory savings can be carried over to animated characters.

Note that some of these issues could be overcome at the cost of lower compression ratios by the use of lower subdivision levels in mesh regions that undergo strong non-rigid deformations. We point this out below in Section 8.

We propose a novel approach which addresses silhouette, normal and tangent frame artefacts for animated DMMs. Our approach uses barycentric interpolation of animation matrices defined for each vertex of the DMM base mesh. The resulting interpolated matrices are then used to transform the DMMs' micro-poly vertices and normals as defined in the original fixed pose. This has the advantage that both the displaced vertices and normals are locally consistent with the encoded DMM representation. For rasterization, interpolation of animation matrices can be efficiently implemented by e.g., mesh shaders [Mic19b]. Even if these interpolated matrices are not used to deform DMM geometry, they can still be used to construct a higher quality normal and tangent frame per micro-triangle.

In the context of ray tracing, tight bounding volumes are required for efficiently detecting intersections between a ray and DMM sub-regions. Our approach allows for efficient bounding of DMM sub-regions by evaluating two quadratic triangular Bézier patches [Far02]. The first Bézier patch provides position data while the second Bézier patch generates normals which then are scaled by the precomputed DMM displacements. Together this data is used to compute a tight bounding volume of the sub-region's convex hull. This sub-region bounding is applied during hierarchical patch subdivision, culling sub-regions which a ray does not intersect. Our bounding approach allows for an efficient tessellation-free ray-DMM intersection test.

2. Related Work

Displaced subdivision surfaces [LMH00] have been a popular approach for adding geometric detail to a coarse base mesh representation. Subdivision surfaces can evaluated either directly or using hierarchical subdivision. The results are on the limit surface and can be displaced along the limit surface normals at the given points. Expressing highly-detailed geometry by applying a set of displacements on a hierarchically refined surface over a coarse base mesh allows for a compact and memory-efficient representation. Our approach for animated DMMs follows a similar path of using only one single set of displacements, the precomputed DMM displacements, even in the context of animations.

Besides hierarchical encoding of displacements, Smith et al. [SSHI00] presented an approach very similar to DMMs as proposed by Maggiordomo et al. [MMT23]. Their work even discusses animating displaced meshes. They do not describe any geometric artefacts seen during animation which can most likely be attributed to a limited set of test cases.

Our work uses quadratic Bézier triangles to efficiently bound animated DMMs. Boubekeur et al. [BA08] introduced quadratic triangular patches for smoothing triangular meshes. Similarly, PN triangles [VPBM01] based on cubic triangular Bézier patches use (quadratic) Bézier triangles to generate smooth normals. In a similar vein, Loop et al. [LSNCn09] use quadratic tangent patches to compute smooth normals. However, we are not aware of a method that uses a second surface patch to compute displacement directions.

Our work focuses on ray tracing animated DMMs but various approaches for ray tracing general displaced meshes have been proposed before. Smits et al. [SSS00] proposed a method for direct ray tracing of displacement mapped meshes but does not handle hierarchical displacement encoding as used for DMMs. Thonat et al. [TBS*21] proposed a tessellation-free ray tracing algorithm for displacement mapped meshes. In their work, displacements are sampled from a displacement texture instead of using a hierarchically encoded displacements and their approach relies on affine arithmetic [dFS04] to bound the displaced micro geometry. DMMs encode displacements hierarchically. They perform repeated 1 : 4 subdivision to introduce new vertices until the desired detail is reached. After each subdivision step, linear predictions of displacements at new vertices are computed. Only correctional deltas between the predicted and measured displacements are stored, using fewer bits at increasing subdivision levels. The use of displacement maps by de Figueiredo et al. [dFS04] does not allow for the same compression ratios at the same surface quality that DMM encoding achieves. As the highly-efficient memory representation is one of the key attributes of DMMs, we only use some key ideas of the tessellation-free ray tracing algorithm described by Thonat et al. [TBS*21].

© 2024 Eurographics - The European Association
for Computer Graphics and John Wiley & Sons Ltd.

```
float4 SkinPosAndNorm(float3 PosIn,
                      inout float3 NormInOut,
                      float4 Weights,
                      uint4 Bones)
{
 matrix M_accum = {0};

 for (int i = 0; i < 4; ++i)
  M_accum += Weights[i] * FetchBoneXfm(Bones[i]);

 NormInOut = mul(float4(NormInOut,0.0f),M_accum);

 return mul(float4(posIn,1.0f), M_accum);
}
```

Listing 1: *Skinning positions and normals using four bones per vertex. The weighted sum of bone matrices is used to transform the vertex and the normal.*

Munkberg et al. [MHTAM10] introduced an efficient culling algorithm for displaced rectangular Bézier patches that uses oriented bounding boxes to bound the displaced patches.

DMMs [MMT23] are limited by topological constraints, as a DMM over a base triangle can only represent micro-geometry that can be expressed as displacements along the interpolated normals of the base triangle. This means the topological genus of a DMM surface, over one base mesh triangle, is zero. Other lossily compressed micro-geometry representations like Nanite clusters [Bri22] or DGF (dense geometry format) blocks [BBM24] do not have these topological limitations but exhibit a lower degree of compression density. Note that DGF is particularly geared at being consumed by future ray tracing hardware. As each DGF block encapsulates a small mesh and a BVH is built over DGF blocks, BVH build complexity is reduced. The cost of a build is proportional to the number of DGF blocks.

Our approach relies on skinned animations of triangle meshes which are based on LBS animated bone matrices. The actual animation of the bone matrices is typically driven by a quaternion-based interpolation (see, e.g., [Han12]) between key-frames. Each key-frame stores orientation, position and potentially scaling data for all bones use by the animated object.

3. Animating Displaced Micro-Meshes

In general, LBS mesh animation consists of two steps: the first step computes a weighted sum of (typically affine) 4×4 matrices associated with each vertex of the mesh. The second step applies the resulting matrix to transform each vertex and a corresponding normal [SSHI00]. Note that this method can be applied regardless of whether the mesh is a regular mesh, a subdivision surface or DMM base mesh. Also, the number of matrices associated with each vertex is typically small, usually less than four for computer game applications. The weights required for computing the weighted sum are typically stored alongside the vertex. The vertex transformation uses a weighted sum of matrices as illustrated in the HLSL code snippet in Listing 1.

Note that the example snippet uses four bones per vertex, while each bone contributes a matrix. The function `FetchBoneXfm` returns the current matrix for bone i for the given vertex. Some implementations do not accumulate a weighted sum of matrices. Instead, for each mesh vertex, they iterate over the matrices, transform the vertex by each matrix and compute a sum of weighted vertices.

In the general case, the code computes an accumulated matrix $M_a(v) = \sum_{n=0}^{N} w_n(v) M_n(v)$ for vertex v. The upper 3×3 part of the matrix is often used to encode rotation data only. It is assumed to be 'sufficiently' orthonormal, and can thus also be used to transform a shading normal. Note that if the upper 3×3 is not sufficiently orthonormal, e.g., if the bone animation applies key-frame dependent scaling, the transpose of its inverse needs to be used to transform shading normals.

We now turn to describing the current state-of-the-art to use LBS on animated objects comprised of DMMs. As there are no explicit bone and weight assignments for the vertices of micro-triangles created during DMM surface reconstruction, the method of choice is to only apply LBS to the vertices and normals of the base mesh. Micro-triangles then get generated during subdivision from the skinned/animated base mesh positions and base mesh normals, and the pre-computed displacements are applied on the interpolation between the animated base mesh normals.

A DMM base mesh triangle consists of three base vertices v_0, v_1 and v_2, and three corresponding (per vertex) base mesh normals n_0, n_1 and n_2. The three accumulated matrices, one per base mesh vertex, are computed:

$$
\begin{aligned}
M_{a_0} &= \sum_{n=0}^{N} w_n(v_0) M_n(v_0) \\
M_{a_1} &= \sum_{n=0}^{N} w_n(v_1) M_n(v_1) \\
M_{a_2} &= \sum_{n=0}^{N} w_n(v_2) M_n(v_2)
\end{aligned} \tag{1}
$$

We apply the accumulated matrices to the three base mesh vertices, which results in $v_0\prime$, $v_1\prime$ and $v_2\prime$:

$$
\begin{aligned}
v_0\prime &= v_0 \cdot M_{a_0} \\
v_1\prime &= v_1 \cdot M_{a_1} \\
v_2\prime &= v_2 \cdot M_{a_2}
\end{aligned} \tag{2}
$$

The same accumulated matrices are used to transform the three positions p_{o_0}, p_{o_1} and p_{o_2}:

$$
\begin{aligned}
p_{o_0} &= (v_0 + n_0) \cdot M_{a_0} \\
p_{o_1} &= (v_1 + n_1) \cdot M_{a_1} \\
p_{o_2} &= (v_2 + n_2) \cdot M_{a_2}
\end{aligned} \tag{3}
$$

These (p_{o_0}, p_{o_1} and p_{o_2}) are generated by displacing the base mesh vertices along the base mesh normals before transforming them. The next step computes the transformed normals $n_0\prime$, $n_1\prime$ and $n_2\prime$, as shown in Equation 4. Note that normal vectors n_0, n_1 and n_2 are four component vectors that have zero in the fourth component and that Equation 4 also is valid if the 3×3 part of M_{a_0}, M_{a_1} and M_{a_2} is not orthogonal.

© 2024 Eurographics - The European Association
for Computer Graphics and John Wiley & Sons Ltd.

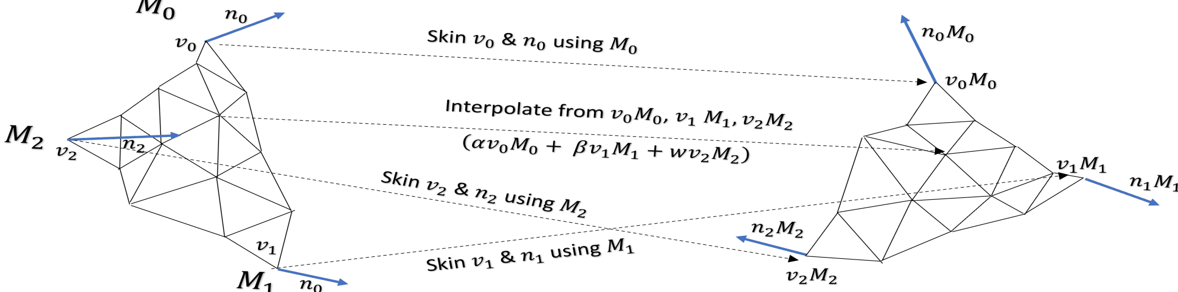


Figure 4: *The current state-of-the-art is to skin the DMM base mesh positions and normals and to interpolate between these results for micro-vertices using their barycentric coordinates* α, β *and,* γ.

$$\begin{aligned} n_0\prime &= p_{o_1} - v_0\prime = n_0 \cdot M_{a_0} \\ n_1\prime &= p_{o_1} - v_1\prime = n_1 \cdot M_{a_1} \\ n_2\prime &= p_{o_2} - v_2\prime = n_2 \cdot M_{a_2} \end{aligned} \tag{4}$$

A DMM is comprised of micro-triangles. In the following we define a DMM micro-vertex as a vertex of any DMM micro-triangle. Every DMM micro-vertex (produced by hierarchical subdivision) at any given subdivision level can be computed by, in a first step, interpolating between transformed base mesh positions and normals. Then, in a second step, the resulting interpolated position is displaced along the interpolated normal. Given barycentric coordinates α, β, $\gamma = 1 - \alpha - \beta$, and a scalar displacement $d(\alpha,\beta,\gamma)$, the interpolated position of the final animated micro-vertex $P_{\text{anim}_{\text{old}}}$ is given by:

$$\begin{aligned} P_{\text{anim}_{\text{old}}} &= \alpha \cdot v_0\prime + \beta \cdot v_1\prime + \gamma \cdot v_2\prime + d(\alpha,\beta,\gamma) \cdot (\alpha \cdot n_0\prime + \beta \cdot n_1\prime + \gamma \cdot n_2\prime) \\ &= (\alpha \cdot v_0 \cdot M_{a_0} + \beta \cdot v_1 \cdot M_{a_1} + \gamma \cdot v_2 \cdot M_{a_2}) + \\ &\quad d(\alpha,\beta,\gamma) \cdot (\alpha \cdot n_0 \cdot M_{a_0} + \beta \cdot n_1 \cdot M_{a_1} + \gamma \cdot n_2 \cdot M_{a_2}) \end{aligned} \tag{5}$$

Note that the original DMM definition adds base offsets and biases, but these are ignored here for illustration purposes. Equation 5 shows that interpolated normals do not get normalized in the context of DMMs. There is also, in the context of micro-vertex compututation, no need to use an the inverse transposed of the skinning matrix to transform the normal. This allows the tessellation-free algorithm for ray tracing standard DMMs to bound the micro-geometry in a nested series of triangular bilinear prismoids. We describe this in more detail in Section 6 and Section 7. Normalization does not change the direction of an interpolated normal but influences the magnitude of displacements only. Thonat et al. [TBS*21] use normalized interpolated normals and thus need to employ a more complicated algorithm and more complex computations for bounding volumes using *affine arithmetic* [dFS04]. We strive to stay as close as possible to standard DMMs and do not normalize interpolated normals. This allows us to directly consume the compressed data blocks that DMMs use without any change.

As described in Section 1, using LBS on the DMM base mesh positions and normals only, can result in artefacts during animation as shown in Figure 1b. These artefacts can affect internal and external silhouettes of animated objects. The use of the resulting micro-triangles to compute tangent frames or micro-triangle normals can result in shading artefacts as shown in Figure 1b.

Before we describe our new approach to skinning animated DMMs we summarize the current state-of-the-art in Figure 4

Now we describe our new approach to skin an animated DMM, as shown in Figure 5. Our approach follows a novel and different path:

1. We reconstruct the micro-triangles of the DMM in the rigging pose they were created from. This produces an almost artefact-free reconstruction of the original high detail object.

2. We implicitly and automatically assign bone matrices and weights to micro-vertices introduced at predefined barycentric coordinates during DMM surface reconstruction. Instead of using barycentric interpolation of the transformed DMM base vertices and normals at the barycentric coordinates of a micro-vertex, we use a barycentric interpolation of the animated accumulated transformation matrices (see Equation 6) M_{a_0}, M_{a_1} and M_{a_2} from Equation 1.

3. We apply the resulting interpolated matrix to the reconstructed micro-vertex in the rigging pose's base mesh vertices and normals.

In general, using interpolated animation matrices is not advisable when matrices contain rotations. In the context of skinning for video game characters this is tolerable because of two reasons: first, the weights and matrices have been crafted by human animators for a range of relevant motions and second, the matrices, which are typically derived from quaternion interpolation [Han12], are sampled with small intermediate time intervals. In most cases the three vertices of a DMM base mesh share a common set of matrices with similar weights. As the skinning matrices are considered well behaved and valid for animating the DMM base mesh, we can assume that the interpolated matrix over the base triangle domain is valid as well. In Section 4 we will show that the interpolation can be considered as the sweeping of a quadratic surface.

Given the three accumulated matrices M_{a_0}, M_{a_1} and M_{a_2} for the three base mesh vertices v_0, v_1 and v_2 (see Equation 1), the interpolated matrix $M_i(\alpha,\beta,\gamma)$ is computed by:

© 2024 Eurographics - The European Association
for Computer Graphics and John Wiley & Sons Ltd.

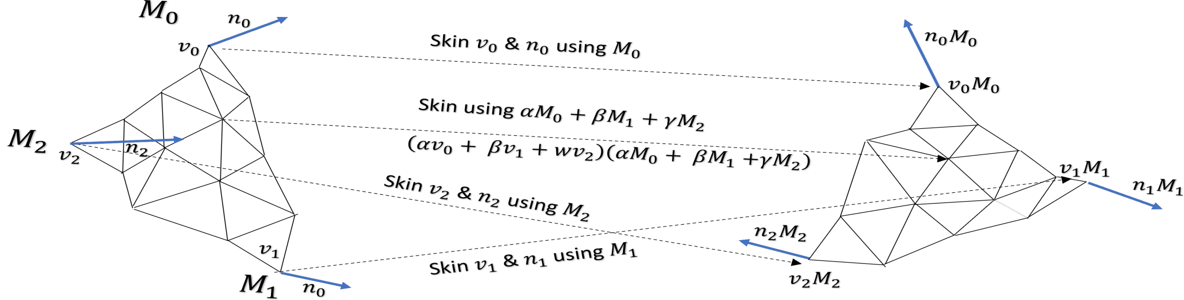


Figure 5: *Our new method interpolates skinning matrices and skins each micro-vertex using their barycentric coordinates* α, β *and,* γ *to create a unique skining matrix.*

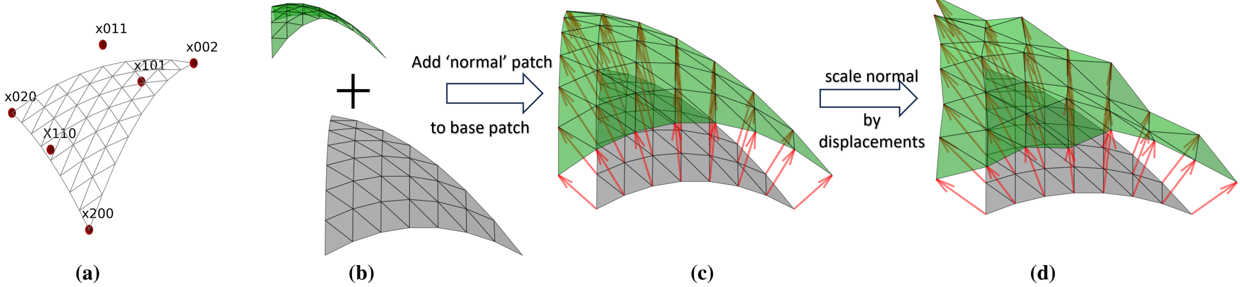


Figure 6: *(a) A degree-two triangular Bézier patch and it's control points* x_{200}, x_{020}, x_{002}, x_{110}, x_{011} *and* x_{101}. *(b) Grey base surface patch and green normals patch. (c) Sum of base surface patch (grey) and normals (red) produces the (green) surface patch. (d) Scaling normals with (scalar) displacements results in a modified displaced green patch.*

$$M_i(\alpha,\beta,\gamma) = \alpha \cdot M_{a_0} + \beta \cdot M_{a_1} + \gamma \cdot M_{a_2} \tag{6}$$

The final vertex $P_{\text{anim}_{\text{new}}}$ is now derived by applying $M_i(\alpha,\beta,\gamma)$ to the interpolated and displaced vertex of the original base mesh (see Equation 7). This method makes sure that the reconstructed micro-vertex is valid by using the original DMM algorithm to compute its position.

$$\begin{aligned} P_{\text{anim}_{\text{new}}}(\alpha,\beta,\gamma) =& (\alpha \cdot v_0 + \beta \cdot v_1 + \gamma \cdot v_2) \cdot M_i(\alpha,\beta,\gamma) + \\ & d(\alpha,\beta,\gamma) \cdot (\alpha \cdot n_0 + \beta \cdot n_1 + \gamma \cdot n_2) \cdot M_i(\alpha,\beta,\gamma) \end{aligned} \tag{7}$$

It is possible to rewrite Equation 7 to show that animated positions in our new scheme can be written as the sum of the two surfaces $P_{\text{anim_pos}}$ and $P_{\text{anim_normal}}$ (Equation 8). Relying on matrix interpolation for computing the final DMM micro-vertex provides consistent results, thereby removing almost all artefacts on silhouettes as seen before (see Figure 1d).

$$P_{\text{anim}_{\text{new}}}(\alpha,\beta,\gamma) = P_{\text{anim_pos}}(\alpha,\beta,\gamma) + d(\alpha,\beta,\gamma) \cdot P_{\text{anim_normal}}(\alpha,\beta,\gamma) \tag{8}$$

As shown in in Figure 1c it is possible to use our method to compute improved micro-triangle tangent frames and normals but to not affect the geometry of the animated DMM. This approach does not lead to improved silhouettes but has the benefit of using an existing DMM ray tracing pipeline with improved tangent frames and normals.

4. A Bézier form for Animated Displaced Micro Meshes

For rasterization, Equation 7 can be easily implemented using e.g., programmable mesh shaders [Mic19b]. Ray tracing DMMs animted by our new scheme, however, requires efficient computation of three-dimensional bounding volumes of DMM sub-regions given a two-dimensional range in the barycentric parameter domain. In this section, we analyze how to rewrite Equation 8 in order to efficiently obtain tight bounding volumes for animated DMMs and DMM sub-regions. In the following we define an *animated DMM patch* as the animated micro-poly surface that a DMM creates when it is skinned or animated by a barycentric blend of three skinning or animation matrices. An *animated DMM patch* is defined by three base mesh vertices, normals and a hierarchy of displacements.

Expanding the first part of Equation 8 (interpolation of animated base mesh positions) yields:

$$\begin{aligned} P_{\text{anim_pos}}(\alpha,\beta,\gamma) =& \alpha^2 \cdot v_0 \cdot M_{a_0} + \beta^2 \cdot v_1 \cdot M_{a_1} + \gamma^2 \cdot v_2 \cdot M_{a_2} + \\ & \alpha \cdot \gamma \cdot (v_2 \cdot M_{a_0} + v_0 \cdot M_{a_2}) + \\ & \alpha \cdot \beta \cdot (v_1 \cdot M_{a_0} + v_0 \cdot M_{a_1}) + \\ & \beta \cdot \gamma \cdot (v_2 \cdot M_{a_1} + v_1 \cdot M_{a_2}) \end{aligned} \tag{9}$$

© 2024 Eurographics - The European Association
for Computer Graphics and John Wiley & Sons Ltd.

A quadratic triangular Bézier patch can be written as shown in Equation 10. Its six control points x_{200}, x_{020}, x_{002}, x_{110}, x_{011} and x_{101} are placed along the border curves (see Figure 6a).

$$\begin{aligned} P(\alpha,\beta,\gamma) = & \alpha^2 \cdot x_{200} + \beta^2 \cdot x_{020} + \gamma^2 \cdot x_{002} + \\ & 2\cdot\alpha\cdot\beta\cdot x_{110} + 2\cdot\beta\cdot\gamma\cdot x_{011} + \\ & 2\cdot\alpha\cdot\gamma\cdot x_{101} \end{aligned} \tag{10}$$

Comparing Equations 9 and 10 shows that we can write Equation 9 as a quadratic triangular Bézier patch. The control points of this patch are given by:

$$\begin{aligned} x_{200} &= v_0 \cdot M_{a_0} \\ x_{020} &= v_1 \cdot M_{a_1} \\ x_{002} &= v_2 \cdot M_{a_2} \\ x_{110} &= \frac{v_1 \cdot M_{a_0} + v_0 \cdot M_{a_1}}{2} \\ x_{011} &= \frac{v_2 \cdot M_{a_1} + v_1 \cdot M_{a_2}}{2} \\ x_{101} &= \frac{v_2 \cdot M_{a_0} + v_0 \cdot M_{a_2}}{2} \end{aligned} \tag{11}$$

Relying on the properties of triangular Bézier patches [Far02] allows for bounding a patch by looking solely at the convex hull spawned by the patch's control points; any bounding volume which bounds the control points also bounds the patch itself.

Equations 10 and 11 can be used in the same way to compute the interpolated normal, just replacing v_i by n_i. As we have seen in Equation 8, we can express the animated DMM by evaluating two quadratic triangular Bézier patches $P_{\text{anim_pos}}(\alpha,\beta,\gamma)$ and $P_{\text{anim_normal}}(\alpha,\beta,\gamma)$.

Note that only using Equation 9 without any displacements already acts like as a smoothing operator on the base mesh similar to [BA08] as shown in Figure 7.

(a) **(b)** **(c)**

Figure 7: *(a) A close-up of the left knee of the frog model's DMM base mesh during animation without DMM displacements. (b) Same close-up but micro-triangle vertices are computed using Equation 9 - again without DMM displacements. (c) Same close-up but micro-triangle vertices are computed using Equation 7 which uses the DMM displacements.*

5. Computing Efficient Bounding Volumes

Computing the DMM displacements includes hierarchical subdivision of each the base mesh triangle in a 4 : 1 fashion. As shown in Equation 8, an animated DMM can be written as the sum of one triangular Bézier patch and a displacement scaled second triangular Bézier patch. Using Equation 11 and the minimum and the maximum displacement for a given DMM base mesh triangle, two new surfaces $P_{\min}$ and $P_{\max}$ can be defined (see Figure 8):

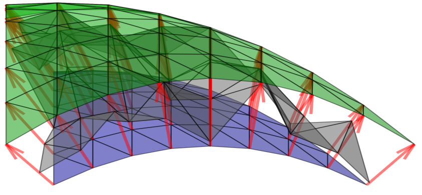


Figure 8: *(Blue) minimum patch $P_{\min}$ (Equation 12), (green) maximum patch $P_{\max}$ (Equation 12) encapsulate the (grey) displaced animated DMM P_{anim} over one base mesh triangle (Equation 10). Red arrows illustrate base position displacement direction by P_{normal}.*

$$P_{min/max} = P_{pos_anim}(\alpha,\beta,\gamma) + d_{min/max} \cdot P_{normal_anim}(\alpha,\beta,\gamma) \tag{12}$$

All possible micro vertices of an animated DMM are guaranteed to be contained within the curved prism spawned by P_{min} and P_{max}. Due to Equation 11, the convex hull of the combined 12 control points (Equation 13) of P_{min} and P_{max} (6 control points each) conservatively bounds the curved prism and therefore bounds the animated DMM itself.

$$\begin{aligned} x_{200_{min/max}} &= (v_0 + d_{min/max} \cdot n_0) \cdot M_{a_0} \\ x_{020_{min/max}} &= (v_1 + d_{min/max} \cdot n_1) \cdot M_{a_1} \\ x_{002_{min/max}} &= (v_2 + d_{min/max} \cdot n_2) \cdot M_{a_2} \\ x_{110_{min/max}} &= \frac{(v_1 + d_{min/max} \cdot n_1) \cdot M_{a_0} + (v_0 + d_{min/max} \cdot n_0) \cdot M_{a_1}}{2} \\ x_{011_{min/max}} &= \frac{(v_2 + d_{min/max} \cdot n_2) \cdot M_{a_1} + (v_1 + d_{min/max} \cdot n_1) \cdot M_{a_2}}{2} \\ x_{101_{min/max}} &= \frac{(v_2 + d_{min/max} \cdot n_2) \cdot M_{a_0} + (v_0 + d_{min/max} \cdot n_0) \cdot M_{a_2}}{2} \end{aligned} \tag{13}$$

We chose to construct an oriented bounding box (*OBB*) from the control points from Equation 13. This bounding box is used in the following as a bounding volume for ray intersection testing. We construct our OBB in a similar way as described in [MHTAM10] but adapt their method to triangular Bézier patches.

6. Ray Tracing Animated Displaced Micro Meshes

Similar to [TBS*21] our approach for ray tracing an animated DMM patch relies on traversing an implicit quad tree of bounding volumes generated by recursive DMM patch subdivision. An animated DMM patch subdivision step produces four sub-patches, hence the quad-tree structure. The bounding volumes of these sub-DMMs are computed on-the-fly without any explicit tessellation involved. In the following we will provide a more detailed description of all steps required.

© 2024 Eurographics - The European Association
for Computer Graphics and John Wiley & Sons Ltd.

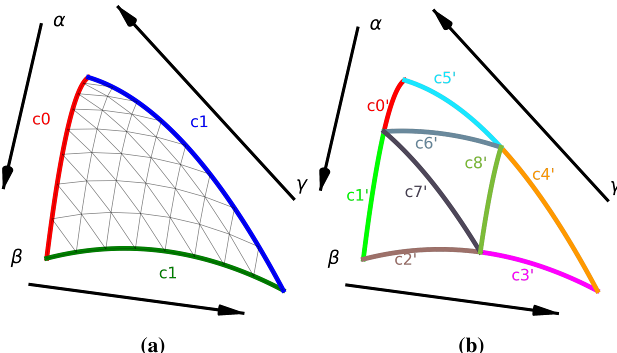


Figure 9: *(a) Quadratic Bézier triangle with three border curves. (b) A 1 : 4 subdivision step produces four children, each one a degree-two Bézier triangle, resulting in nine border curves total.*

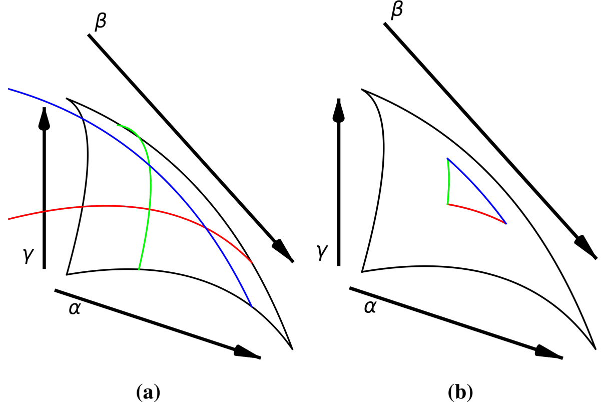


Figure 10: *(a) The three (red, green and blue) unclipped border curves for an arbitrary subpatch. (b) The clipped curves that form the actual subpatch.*

If a ray is found to intersect an animated DMM patch's (oriented) bounding box, its four new sub-patches (see Figure 9) get computed. For each sub-patch, an oriented bounding box is created. Then each sub-patch is processed recursively. We can optionally sort the boxes based on the distance of their center to the starting point of the ray to facilitate a front-to-back inspired traversal order. Note that other and more elaborate sorting algorithms could be used. The recursion continues until the subdivision level is deep enough to switch to micro-triangle testing. At that point ray-triangle intersection tests are used to test the ray against the (animated) micro-triangles. As our method for bounding DMM patches relies on quadratic Bézier patches, we need to be able to compute Bézier sub-patches at each subdivision level. Instead of keeping a stack of patches for recursive splitting, we compute the required Bézier subpatches and their border curves directly from the top level patches.

Figure 10a shows the three unclipped degree-two border curves that form the borders of a subpatch inside a degree-two triangular Bézier patch. Figure 10b shows the clipped curves that fully define the subpatch. For each of the three border curves we start by computing the coefficients of the full curves as shown in Figure 10a. Next we offset the starting point of each curve and reparameterize it to only run the length of a border curve for the required subdivision level. In a final step we convert the three curves, each represented by three polynomial coefficients a, b and c, back to Bézier form. Rewriting a quadratic polynomial at^2+bt+c in Bézier form $(1-t)^2cp_0+2(1-t)tp_1+t^2cp_2$ yields $cp_0=c$, $cp_1=0.5b+c$ and $cp_2=a+b+c$. This allows us to compute the control points of the subpatch. Note that it is also possible to compute Bézier control points for the curves in Figure 10a and to then use blossom notation (see [Far02]) to directly compute the Bézier control points of the clipped curves.

We can now summarize the tessellation-free ray tracing algorithm in Algorithm 1. The entry point to raytrace an animated DMM patch is RAYTRACEANIMATEDDMM. This function takes a ray, an array of three base mesh vertices, an array of three normals and an array of three accumulated animation matrices as inputs. First it computes the control points of top-level position and normal patch P_{normal} and P_{position} from Equation 10. It then utilizes a precomputed hierarchy of minimum and maximum displacements (see Figure 11) to create the control points of $P_{\min}$ and $P_{\max}$ (see Equations 12 and 13) and computes a bounding box for the whole DMM. Next it iterates over the four sub-patches of P_{normal} and P_{position}. Using the precomputed hierarchy of minimum and maximum displacements, a bounding box is created for each subpatch. Here each subpatch is computed directly from the top-level patches. Bounding boxes are sorted to prioritize bounding boxes closer to the origin of the ray. Then RAYTRACESUBPATCH is called to recursively handle the intersection of each animated DMM subpatch with the ray. The function *computeSubPatch* utilizes a running count of visited subpatches and combines this with the index of the current subpatch and its subdivision levels to compute the subpatch from Figure 10b.

Algorithm 1 shows that the algorithm structure for ray tracing animated DMMs using our interpolated matrix approach is roughly similar to the structure of a tessellation-free algorithm for ray tracing static DMMs (see Algorithm 2). Algorithm 2 also relies on a precomputed hierarchy of minimum and maximum displacements and recursively splits a static DMM base triangle using a 1 : 4 subdivision scheme (Figures 12b and 12c). In the static case, the DMM is within the volume of a triangular bilinear prismoid [MMT23]

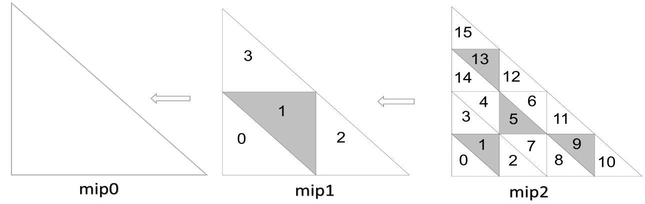
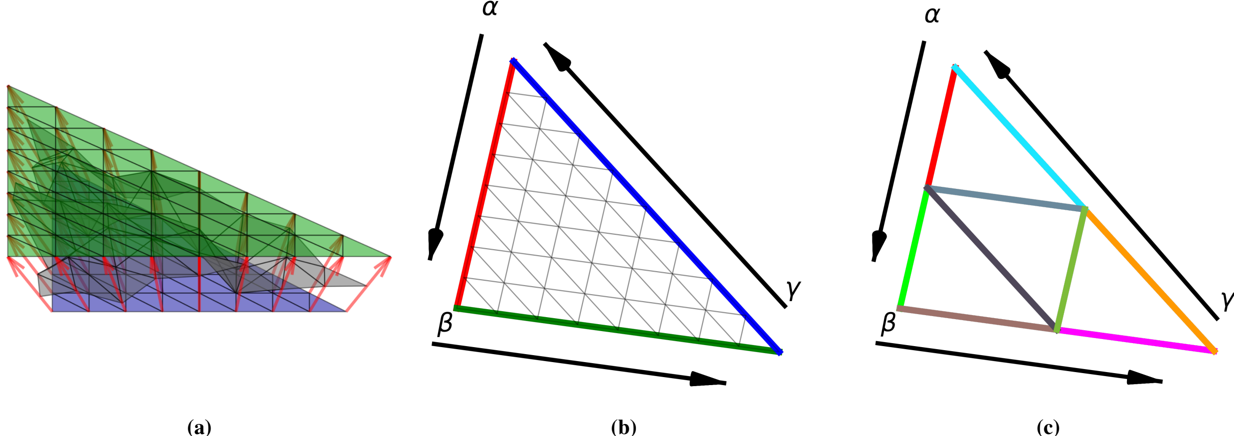


Figure 11: *Building a triangular min/max displacement hierarchy starting at mip2 at the right. The mip2 min/max displacements from subtriangles with IDs $0-3$, $4-7$, $8-11$ and $12-15$ get combined for subtriangles with IDs 0, 1, 2 and 3 at mip1. In the next step min/max displacements from subtriangles with IDs $0-3$ at mip1 get combined to compute the min/max displacements of a base mesh triangle at mip0.*

© 2024 Eurographics - The European Association
for Computer Graphics and John Wiley & Sons Ltd.



(a) (b) (c)

Figure 12: *(a) Triangular bilinear prismoid that bounds a static DMM. (b) Static DMM base triangle along with the three barycentric axis α, β and γ. (c) Base triangle is split into four sub-triangles.*

(see Figure 12a). After splitting a uv-space triangle 1 : 4 (see Figures 12b and 12c), the algorithm recursively constructs and tests each triangular bilinear sub-prismoid. Note that both algorithms, for simplicity, omit details about fetching and applying additive hierarchical DMM displacements.

7. Results and Discussion

For our evaluation, we used a prototype DXR [Mic20] application which implements Algorithm 1 as an intersection shader [Mic19a]. All tests were conducted on an AMD® Radeon™ 7900 XT GPU and Windows 11.

Our approach for ray tracing animated DMMs object consists of three steps. In the first step, a GPU compute pass stores the accumulated weighted skinning matrices (as defined by the current animation pose - see Equation 1) for each object (comprised of animated DMMs) and all its DMM patches to a GPU buffer. The same GPU pass computes the AABB of each DMM patch's twelve control points (minimum and maximum patch, see Figure 8) and stores these AABBs to a separate GPU buffer. In the second step, one bottom-level acceleration structure (BLAS) for each object comprised of animated DMMs is built. Each BLAS contains all the patch AABBs of its object. Next, a top-level acceleration structure over the objects' BLAS structures is built. In the final step, rays are generated and dispatched to render the animated DMM model. Note that a full BLAS rebuild is not always necessary, instead a faster BLAS refit is often sufficient for animated game assets.

We evaluate three animated objects (see Figure 13) with varying complexity and animations. Our approach (bottom row) in Figure 13 does not show the artefacts seen in regular animated DMMs (middle row). For performance comparison, we have implemented a tessellation-free ray tracing algorithm (see Algorithm 2) for static DMMs. This approach uses a 1 : 4 subdivision scheme on triangular prisms and is used as the performance baseline. It follows a flow that is similar to Algorithm 1 but instead of subdividing degree-two triangular Bézier patches it uses planar subdivision. Starting at a base mesh triangle, a maximum bounding triangle (see green triangle in Figure 12a) and a minimum bounding triangle (see blue

	Frog	Hell Hound	Octopus
	Subdivision Level=3		
Our Approach	2.75	6.61	4.05
Static DMM	1.17	2.55	1.84
	Subdivision Level=4		
Our Approach	3.84	8.71	6.20
Static DMM	1.56	3.80	2.61

Table 1: *Rendering time (ms) comparison between our approach and ray tracing static DMMs (primary rays only, full HD resolution) for subdivision levels 3 and 4. Due to higher algorithmic complexity and register pressure, our approach has* $\sim 2.3-2.5\times$ *higher ray tracing costs than regular ray tracing of static DMMs.*

triangle in the Figure 12a) are computed. If the ray intersects the bounding box of the triangular prism, defined by the minimum and the maximum bounding triangle, the triangle is subdivided into four sub-triangles and four sub-prisms. These sub-prisms get recursively tested until the algorithm reaches the level of micro-triangles. Note, that in our algorithms (Algorithm 1 and 2) we, for brevity's sake, ignore the multi-resolution nature of- and watertightness concerns between neighboring DMMs with non-uniform subdivisions levels.

The accumulated skinning matrices at each vertex of the DMM base mesh are only needed when we compute control points of quadratic patches from Equation 10. We need the matrices during BLAS construction to compute the control points of the minimum and maximum Bézier patches for the animated DMM over each base mesh triangle (see Figure 8). These control points are used to compute the bounding volumes (AABBs in our implementation) of each animated DMM patch. Whenever we detect that a ray enters such a bounding volume, the matrices are used to compute the Bézier control points as shown in Algorithm 1. In our prototype implementation, an intersection shader gets invoked when the AABB of a DMM over a base triangle is hit. The intersection shader then recomputes the accumulated skinning matrices for the three base triangle vertices. The final interpolated matrix is then used for Algorithm 1. This means that we do not need to store accumulated skinning matrices for the whole frame, deferring it until BLAS construction. As we use a single AABB to bound each animated DMM

© 2024 Eurographics - The European Association
for Computer Graphics and John Wiley & Sons Ltd.

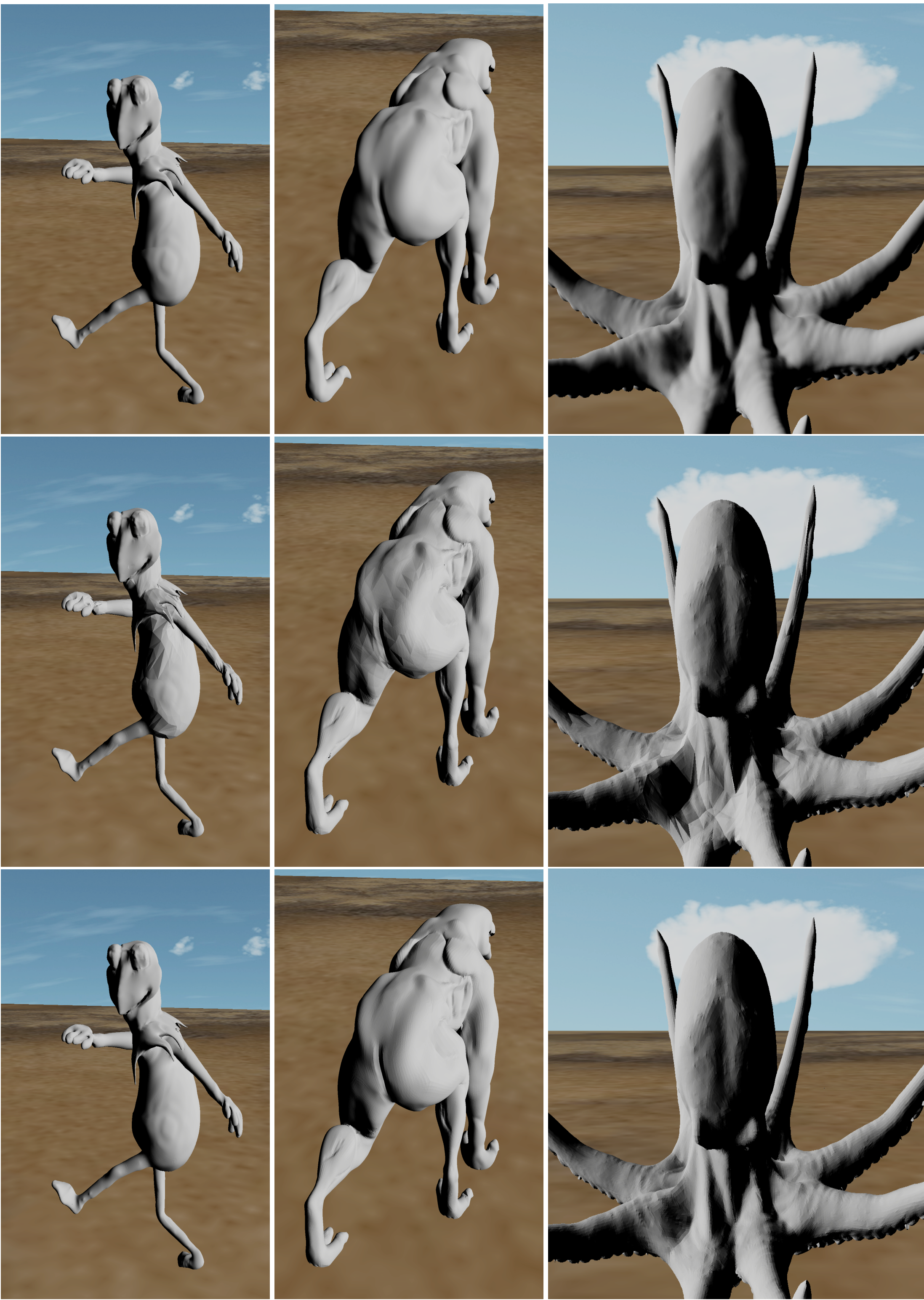


Figure 13: *(Top row) Original high resolution models: frog (∼400k triangles), hell hound (∼362k triangles), and octopus (∼153k triangles) animated through* skinning. *(Middle row) Standard animated displaced micro meshes: frog (∼6k base triangles), hell hound (∼5.7k base triangles), and octopus (∼5.9k base triangles) show geometry and normal artefacts when animated. (Bottom row) Our interpolated matrix approach shows virtually no artefacts.*

© 2024 Eurographics - The European Association
for Computer Graphics and John Wiley & Sons Ltd.

base triangle, BLAS construction and refit times are roughly proportional to the number of triangles in the DMM base mesh.

Table 1 shows a rendering time comparison between our approach (Algorithm 1) and ray tracing static DMMs (Algorithm 2). Due to the higher algorithmic complexity and higher GPU shader register pressure, our approach is $\sim 2.3-2.5\times$ slower

8. Conclusion and Future Work

Our proposed approach removes geometric and lighting artefacts for animated DMMs by interpolating (vertex-based) transformation matrices first and using the result to transform DMM vertex and normal data. This retains the main benefits of DMMs in the context of animation: their compact representation and low BVH construction complexity. Our current implementation of the ray DMM intersection algorithm (see Algorithm 1) is $\sim 2.3-2.5\times$ slower than a similar implementation for static DMMs (see Algorithm 2).

Skinned objects can have regions that deform mostly rigidly. For these regions it may be beneficial to only animate (skin) base mesh vertices and normals and to use Algorithm 2 to test for ray intersections. In the future we would like to investigate how to best handle animated objects that are ray traced by a combination of Algorithm 1 and Algorithm 2 depending on how different regions deform during animation. Note that static level of detail across an object comprised of DMMs is supported by Algorithm 1 and 2. Watertightness between DMMs with non-uniform subdivision levels is currently not supported and is planned as future work. Along with support for non-uniform subdivision we'd further investigate how to tailor mesh simplification during DMM construction to generate a DMM base mesh with a higher triangle count in regions that undergo strong deformations in a given animation sequence.

Acknowledgements

The frog, the hell hound and the octopus model are courtesy of the respective artists on SketchFab.com.

References

[BA08] Boubekeur T., Alexa M.: Phong tessellation. In *ACM SIGGRAPH Asia 2008 Papers* (New York, NY, USA, 2008), SIGGRAPH Asia '08. doi:10.1145/1457515.1409094.

[BBM24] Barczak J., Benthin C., McAllister D.: Dgf: A dense, hardware-friendly geometry format for lossily compressing meshlets with arbitrary topologies. *Proc. ACM Comput. Graph. Interact. Tech. 7*, 3 (aug 2024). URL: https://doi.org/10.1145/3675383, doi:10.1145/3675383.

[Bri22] Brian Karis: Journey to Nanite, 2022. URL: https://www.highperformancegraphics.org/slides22/Journey_to_Nanite.pdf.

[dFS04] de Figueiredo L. H., Stolfi J.: Affine arithmetic: Concepts and applications. *Numerical Algorithms 37* (2004), 147–158. URL: https://api.semanticscholar.org/CorpusID:2431872.

[DR05] DeCoro C., Rusinkiewicz S.: Pose-independent simplification of articulated meshes. In *Proceedings of the 2005 Symposium on Interactive 3D Graphics and Games* (New York, NY, USA, 2005), I3D '05, Association for Computing Machinery, p. 17–24. URL: https://doi.org/10.1145/1053427.1053430, doi:10.1145/1053427.1053430.

[Far02] Farin G. E.: *Curves and surfaces for CAGD: a practical guide*. Morgan Kaufmann, 2002.

[GH97] Garland M., Heckbert P. S.: Surface simplification using quadric error metrics. In *Proceedings of the 24th Annual Conference on Computer Graphics and Interactive Techniques* (USA, 1997), SIGGRAPH '97, ACM Press/Addison-Wesley Publishing Co., p. 209–216. URL: https://doi.org/10.1145/258734.258849, doi:10.1145/258734.258849.

[Gru24] Gruenvogel S.: *Linear Blend Skinning*. 04 2024, pp. 373–398. doi:10.1007/978-3-658-41989-9_10.

[Han12] Hanson A. J.: Quaternion applications. In *SIGGRAPH Asia 2012 Courses* (2012), SA '12. doi:10.1145/2407783.2407794.

[Hop23] Hoppe H.: *Progressive Meshes*, 1 ed. Association for Computing Machinery, New York, NY, USA, 2023. URL: https://doi.org/10.1145/3596711.3596725.

[KCvO07] Kavan L., Collins S., Žára J., O'Sullivan C.: Skinning with dual quaternions. In *Proceedings of the 2007 Symposium on Interactive 3D Graphics and Games* (New York, NY, USA, 2007), I3D '07, Association for Computing Machinery, p. 39–46. URL: https://doi.org/10.1145/1230100.1230107, doi:10.1145/1230100.1230107.

[Kiy24] Kiyavash Kandar: Large Scale GPU-Based Skinning for Vegetation in 'Alan Wake 2', 2024. Game Developers Conference 2024.

[LMH00] Lee A., Moreton H., Hoppe H.: Displaced subdivision surfaces. In *Proceedings of the 27th Annual Conference on Computer Graphics and Interactive Techniques* (USA, 2000), SIGGRAPH '00, ACM Press/Addison-Wesley Publishing Co., p. 85–94. URL: https://doi.org/10.1145/344779.344829, doi:10.1145/344779.344829.

[LSNCn09] Loop C., Schaefer S., Ni T., Castaño I.: Approximating subdivision surfaces with gregory patches for hardware tessellation. *ACM Trans. Graph. 28*, 5 (dec 2009), 1–9. URL: https://doi.org/10.1145/1618452.1618497, doi:10.1145/1618452.1618497.

[LWL*12] Liu Y., Wang Y.-J., Lin Y.-Y., Li L.-Z., Qi C.-Y.: Half edge collapse mesh simplification algorithm based on constrained quadric error metric method (hecmsa-cqem)for model simplification. In *2012 Fifth International Conference on Intelligent Networks and Intelligent Systems* (2012), pp. 1–4. doi:10.1109/ICINIS.2012.32.

[MHTAM10] Munkberg J., Hasselgren J., Toth R., Akenine-Möller T.: Efficient bounding of displaced bézier patches. In *Proceedings of the Conference on High Performance Graphics* (Goslar, DEU, 2010), HPG '10, Eurographics Association, p. 153–162.

[Mic19a] Microsoft: DirectX Raytracing (DXR) Functional Spec, 2019. URL: https://microsoft.github.io/DirectX-Specs/d3d/Raytracing.html.

[Mic19b] Microsoft: DirectX-Specs Mesh Shader, 2019. URL: https://microsoft.github.io/DirectX-Specs/d3d/MeshShader.html.

[Mic20] Microsoft: DirectX Raytracing (DXR) Functional Spec, 2020. URL: https://microsoft.github.io/DirectX-Specs/d3d/Raytracing.html.

[MMT23] Maggiordomo A., Moreton H., Tarini M.: Micro-mesh construction. *ACM Trans. Graph. 42*, 4 (jul 2023). doi:10.1145/3592440.

[MOB*21] Meister D., Ogaki S., Benthin C., Doyle M., Guthe M., Bittner J.: A survey on bounding volume hierarchies for ray tracing. *Computer Graphics Forum 40* (06 2021), 683–712. doi:10.1111/cgf.142662.

[SSHI00] Smith R., Sun W., Hilton A., Illingworth J.: Layered animation using displacement maps. In *Proceedings Computer Animation 2000* (2000), pp. 146–151. doi:10.1109/CA.2000.889072.

© 2024 Eurographics - The European Association
for Computer Graphics and John Wiley & Sons Ltd.

[SSS00] SMITS B., SHIRLEY P., STARK M. M.: Direct ray tracing of displacement mapped triangles. In *Rendering Techniques 2000* (Vienna, 2000), Péroche B., Rushmeier H., (Eds.), Springer Vienna, pp. 307–318.

[TBS*21] THONAT T., BEAUNE F., SUN X., CARR N., BOUBEKEUR T.: Tessellation-free displacement mapping for ray tracing. *ACM Trans. Graph. 40*, 6 (dec 2021). doi:10.1145/3478513.3480535.

[VPBM01] VLACHOS A., PETERS J., BOYD C., MITCHELL J. L.: Curved PN Triangles. In *Proceedings of the 2001 Symposium on Interactive 3D Graphics* (2001), I3D '01, p. 159–166. doi:10.1145/364338.364387.

© 2024 Eurographics - The European Association
for Computer Graphics and John Wiley & Sons Ltd.

Algorithm 1 Tessellation-free ray tracing of animated DMMs

```
 1: function CREATEDMMORIENTEDBOUNDINGBOX(base_vertices[3], base_normals[3], accum_matrices[3], i[0..4], level)
 2:     minD, maxD ← getMinMaxDisplacements(level, i[0..4])          ▷ get min/max displ. from the top of the hierarchy
 3:     pmins, pmaxs ← computeMinMaxControlPoints(base_vertices, base_normals, accum_matrices, minD, maxD)   ▷ Equation 13
 4:     return CreateOrientedBoundingBox(pmins, pmaxs)
 5: end function
 6: function RAYTRACESUBPATCH(ray, bbox, TopLevelPositionPatch, TopLevelNormalPatch, accum_matrices[3], level, i[0..4])
 7:     if intersects(ray, bbox) then
 8:         i[level] ← 0
 9:         while i[level] ≠ 4 do
10:             minD, maxD ← getMinMaxDispl(level, i[0..4])          ▷ get displ. from min/max hierarchy
11:             PositionPatch ← computeSubPatch(TopLevelPosPatch, level, i[0..4])          ▷ Figure 9
12:             NormalPatch ← computeSubPatch(TopLevelNormalPatch, level, i[0..4])          ▷ Figure 9
13:             pmin[0..11] ← computePatchControlPoints(PositionPatch.vs, NormalPatch.vs, accum_matrices, minD)   ▷ Equation 11
14:             pmax[0..11] ← computePatchControlPoints(PositionPatch.vs, NormalPatch.vs, accum_matrices, maxD)   ▷ Equation 11
15:             bbox[i] ← createOrientedBoundingBox(pmin, pmax)
16:             i[level] ← i[level] + 1
17:         end while
18:         idx[0..3] ← sortBoxes(ray, bbox[0..3])          ▷ sort e.g., by distance from ray origin
19:         i[level] ← 0
20:         while i[level] ≠ 4 do
21:             if level ≥ MAX_SUB_DIV then
22:                 raytraceSubPatch(ray, bbox[idx[i]], TopLevelPPatch, TopLevelNPatch, accum_matrices[0..2], level + 1, i[0..4])
23:             else
24:                 raytraceAnimatedMicroTriangle(ray, accum_matrices[0..2], i[0..4], TopLevelPositionPatch, TopLevelNormalPatch)   ▷ Algorithm 3
25:             end if
26:             i[level] ← i[level] + 1
27:         end while
28:     end if
29: end function
30: function RAYTRACEANIMATEDDMM(ray, base_verts[0..2], base_normals[0..2], accum_matrices[0..2] )
31:     i[0..4] ← 0
32:     level ← 0
33:     TopLevelPositionPatch ← computeTopLevelPatchControlPoints(base_verts, accum_matrices)          ▷ Equation 11
34:     TopLevelNormalPatch ← computeTopLevelPatchControlPoints(base_normals, accum_matrices)          ▷ Equation 11
35:     bbox = createDMMOrientedBoundingBox(TopLevelPositionPatch, TopLevelNormalPatch)
36:     if intersects(ray, bbox) then
37:         while i[0] ≠ 4 do
38:             minD, maxD ← getMinMaxDispl(l, i[0..4])          ▷ get min/max displ. from hierarchy
39:             PositionPatch ← computeSubPatch(TopLevelPositionPatch, i[0..4], 0)          ▷ Figure 9
40:             NormalPatch ← computeSubPatch(TopLevelNormalPatch, i[0..4], 0)          ▷ Figure 9
41:             pmin[0..11] ← computePatchControlPoints(PositionPatch, NormalPatch, matrices, minD)          ▷ Equation 11
42:             pmax[0..11] ← computePatchControlPoints(PositionPatch, NormalPatch, matrices, maxD)          ▷ Equation 11
43:             bbox[i] ← createBoundingOrientedBoundingBox(pmin[0..11], pmax[0..11])
44:             i ← i + 1
45:         end while
46:         idx[0..3] ← sortBoxes(ray, bbox[0..3])          ▷ sort e.g., by distance from ray origin
47:         i ← 0
48:         while i ≠ 4 do
49:             raytraceSubPatch(ray, bbox[idx[i]], TopLevelPositionPatch, TopLevelNormalPatch, accum_matrices[0..2], level + 1, i[0..4])
50:             i ← i + 1
51:         end while
52:     end if
53: end function
```

© 2024 Eurographics - The European Association
for Computer Graphics and John Wiley & Sons Ltd.

Algorithm 2 Tessellation-free ray tracing of static DMMs

```
 1: function CREATETOPMOSTPRISM(skinned_base_vertices, skinned_base_normals)
 2:     i[0..4] ← 0
 3:     minD, maxD ← getMinMaxDisplacements(0, i[0..4])                              ▷ min/max displ. from hierarchy
 4:     vmax[0..2] ← computeDisplacedTriangle(skinned_base_vertices, skinned_base_normals, maxD)   ▷ top of prism
 5:     vmin[0..2] ← computeDisplacedTriangle(skinned_base_vertices, skinned_base_normals, minD)   ▷ bottom of prism
 6:     return prism( vmin, vmax )
 7: end function
 8: function RAYTRACESUBPRISM(ray, skinned_base_verts, skinned_base_normals, prism, level, i[0..4])
 9:     minD, maxD ← getMinMaxDisplacements(level, i[0..4])                          ▷ min/max displ. from hierarchy
10:     vmin[0..2] ← computeDisplacedTriangle(prism, minD)                            ▷ top of prism
11:     vmax[0..2] ← computeDisplacedTriangle(prism, maxD)                            ▷ bottom of prism
12:     bbox ← createBoundingBox(vmin, vmax)                                          ▷ AABB or OBB
13:     if intersects(ray, bbox) then
14:         SubPrisms[0..3] ← splitPrism(prism)                                       ▷ 4 sub-prisms
15:         SubPrisms[0..3] ← sortPrisms(ray, SubPrisms[0..3])                        ▷ sort by distance from ray origin
16:         i[level] ← 0
17:         while i[level] ≠ 4 do
18:             if level ≥ MAX_SUB_DIV then
19:                 raytraceSubPrism(ray, skinned_base_verts, skinned_base_normals, SubPrisms[i], level + 1, i[0..4])
20:             else
21:                 raytraceMicroTriangle(ray, skinned_base_verts, skinned_base_normals, i[0..4])   ▷ Algorithm 4
22:             end if
23:             i[level] ← i[level] + 1
24:         end while
25:     end if
26: end function
27: function RAYTRACESTATICDMM(ray, skinned_base_verts, skinned_base_normals )
28:     i[0..4] ← 0
29:     level ← 0
30:     prism ← createTopMostPrism(skinned_base_verts, skinned_base_normals)
31:     bbox = createBoundingBox(prism)                                               ▷ AABB or OBB
32:     if intersects(ray, bbox) then
33:         SubPrism[0..3] ← splitPrism(prism)
34:         SubPrism[0..3] ← sortPrisms(ray, SubPrisms[0..3])                         ▷ sort by distance from ray origin
35:         i[0..4] ← 0
36:         while i[0] ≠ 4 do
37:             raytraceSubPrism(ray, SubPrism[i], level, i[0..4])
38:             i[0] ← i[0] + 1
39:         end while
40:     end if
41: end function
```

© 2024 Eurographics - The European Association
for Computer Graphics and John Wiley & Sons Ltd.

Algorithm 3 Ray-trace an animated micro-triangle

```
1: function RAYTRACEANIMATEDMICROTRIANGLE(ray, i[0..4], TopLevelPositionPatch, TopLevelNormalPatch )
2:     uv[0..2], displacements[0..2] ← computeMicroTriangleUVsandDisplacements(i[0..4])
3:     micro_triangles_vert0 ← computeInterpolatedPosition(TopLevelPositionPatch, uv[0])
4:     micro_triangles_vert1 ← computeInterpolatedPosition(TopLevelPositionPatch, uv[1])
5:     micro_triangles_vert2 ← computeInterpolatedPosition(TopLevelPositionPatch, uv[2])
6:     micro_triangles_normal0 ← computeInterpolatedNormal(TopLevelNormalPatch, uv[0])
7:     micro_triangles_normal1 ← computeInterpolatedNormal(TopLevelNormalPatch, uv[1])
8:     micro_triangles_normal2 ← computeInterpolatedNormal(TopLevelNormalPatch, uv[2])
9:     micro_triangles_vert0 ← micro_triangles_vert0 + displacement[0] * micro_triangles_normal0
10:    micro_triangles_vert1 ← micro_triangles_vert1 + displacement[1] * micro_triangles_normal1
11:    micro_triangles_vert2 ← micro_triangles_vert2 + displacement[2] * micro_triangles_normal2
12:    matrix0 ← computeInterpolatedMatrix(accum_matrices[0..2], uv[0])
13:    matrix1 ← computeInterpolatedMatrix(accum_matrices[0..2], uv[1])
14:    matrix2 ← computeInterpolatedMatrix(accum_matrices[0..2], uv[2])
15:    micro_triangles_vert0 ← micro_triangles_vert0 * matrix[0]
16:    micro_triangles_vert1 ← micro_triangles_vert1 * matrix[1]
17:    micro_triangles_vert2 ← micro_triangles_vert2 * matrix[2]
18:    return raytraceTriangle(ray, micro_triangles_vert0, micro_triangles_vert1, micro_triangles_vert2)
19: end function
```

Algorithm 4 Ray-trace a micro-triangle

```
1: function RAYTRACEMICROTRIANGLE(ray, skinned_base_verts[0..2], skinned_base_normals[0..2], i[0..4] )
2:     uv[0..2], displacements[0..2] ← computeMicroTriangleUVsandDisplacements(i[0..4])
3:     micro_triangles_vert0 ← computeInterpolatedPosition(skinned_base_verts[0..2], uv[0])
4:     micro_triangles_vert1 ← computeInterpolatedPosition(skinned_base_verts[0..2], uv[1])
5:     micro_triangles_vert2 ← computeInterpolatedPosition(skinned_base_verts[0..2], uv[2])
6:     micro_triangles_normal0 ← computeInterpolatedNormal(skinned_base_normals[0..2], uv[0])
7:     micro_triangles_normal1 ← computeInterpolatedNormal(skinned_base_normals[0..2], uv[1])
8:     micro_triangles_normal2 ← computeInterpolatedNormal(skinned_base_normals[0..2], uv[2])
9:     micro_triangles_vert0 ← micro_triangles_vert0 + micro_triangles_normal0 * displacement[0]
10:    micro_triangles_vert1 ← micro_triangles_vert1 + micro_triangles_normal1 * displacement[1]
11:    micro_triangles_vert2 ← micro_triangles_vert2 + micro_triangles_normal2 * displacement[2]
12:    return raytraceTriangle(ray, micro_triangles_vert0, micro_triangles_vert1, micro_triangles_vert2)
13: end function
```

© 2024 Eurographics - The European Association
for Computer Graphics and John Wiley & Sons Ltd.