Efficient Ray Tracing of Micro-Meshes

IN5000: Final Project

Patrick Hibbs

Delft University of Technology

TU Delft

Efficient Ray Tracing of Micro-Meshes

by

Patrick Hibbs

Supervisor:	Ricardo Marroquim
Project Duration:	January, 2025 - October, 2025
Faculty:	EEMCS, TU Delft

TU Delft

Acknowledgements

I would like to express my gratitude to Professor Ricardo Marroquim for his expertise and guidance during my thesis project. I would also like to thank Holger Gruen (AMD) for sending over part of his intersection shader from his work [8], even though it was ultimately not required.

The meshes used in this thesis were all acquired from Three D Scans [13], and converted into micro-meshes using the method of Maggiordomo et al. [16].

Abstract

This thesis presents the design and implementation of a GPU-based ray tracer capable of rendering NVIDIA's micro-meshes. The method operates entirely in two dimensions by projecting both rays and triangles onto a 2D domain. Within this domain, the projected ray traverses the micro-triangles, starting at subdivision level 0. The triangles are then recursively subdivided until the finest subdivision level is reached, at which point the micro-triangle intersected by the original 3D ray is identified. To accelerate traversal, the approach employs height field ray tracing, which restricts subdivision to regions where it is strictly necessary. In addition, the proposed method demonstrates a reduced memory footprint compared to traditional ray tracing, while maintaining acceptable runtime performance. Memory savings of up to approximately 50% are achieved relative to a fully tessellated representation.

Contents

1 Introduction

Micro-meshes [21] are an advanced rendering technique designed to efficiently represent highly detailed geometric surfaces without the overhead of fully tessellated models. Unlike traditional tessellation methods, which require subdividing meshes at run-time, micro meshes enable finer surface detail while maintaining a compact memory footprint. This makes them particularly useful for real-time applications, such as games and interactive simulations.

A foundational technique to improve surface detail is displacement mapping, where a texture-based height field displaces the underlying geometry [14]. Although displacement maps provide a flexible way to introduce detail, they typically rely on tessellation or adaptive subdivision to achieve high fidelity.

NVIDIA introduced micro-meshes as a novel alternative, aiming to optimize geometric complexity using a hierarchical representation [20]. By leveraging precomputed micro triangles, micro-meshes allow efficient ray tracing and level-of-detail management without excessive memory or performance costs.

To construct micro-meshes from traditional polygonal models, specialized algorithms or applications are used to generate a compact representation that preserves the original shape while optimizing storage and traversal efficiency [16, 6, 26].

By enabling high-resolution details with minimal performance impact, micro-meshes represent a significant advancement in real-time rendering, particularly in ray tracing applications. This thesis aims to contribute to this development by providing an efficient method for ray tracing micro-meshes.

2

Related Work

The related work is organized into three primary sections. The first section reviews related work on micro-meshes and surface representations. The second section examines approaches to ray tracing displaced meshes. Finally, the third section discusses an alternative to micro-meshes.

2.1. Micro-meshes and Surface Representation

While micro-meshes in their current form are relatively new, the idea of using a compressed and coarser version of a mesh to encode surface details has existed for a longer time. The first approach was displacement mapping, introduced by Cook in 1984, which allowed fine surface details to be represented by displacing a base surface (see Figure 2.1) [4].

Another step towards efficient surface representation was made with normal meshes [9], which encode high-resolution surfaces as a coarse base mesh augmented with displacement vectors. Unlike the displacement mapping done by Cook, which relies on texture-space displacements, normal meshes store displacements in relation to the local surface, making them well-suited for areas such as rendering and compression.

Curved PN triangles [30] define smooth surfaces over triangular meshes, offering a higher fidelity representation. It works by substituting the regular flat triangles with cubic Bézier triangles (see Figure 2.2). The control points of these Bézier triangles allow for curving triangles, creating a smoother surface.

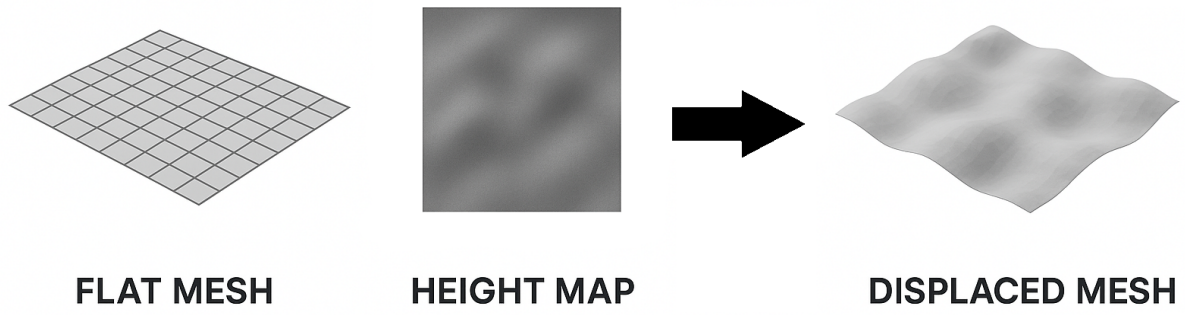


Figure 2.1: Displacement mapping. A base surface is modified according to a displacement texture to produce a detailed displaced surface.

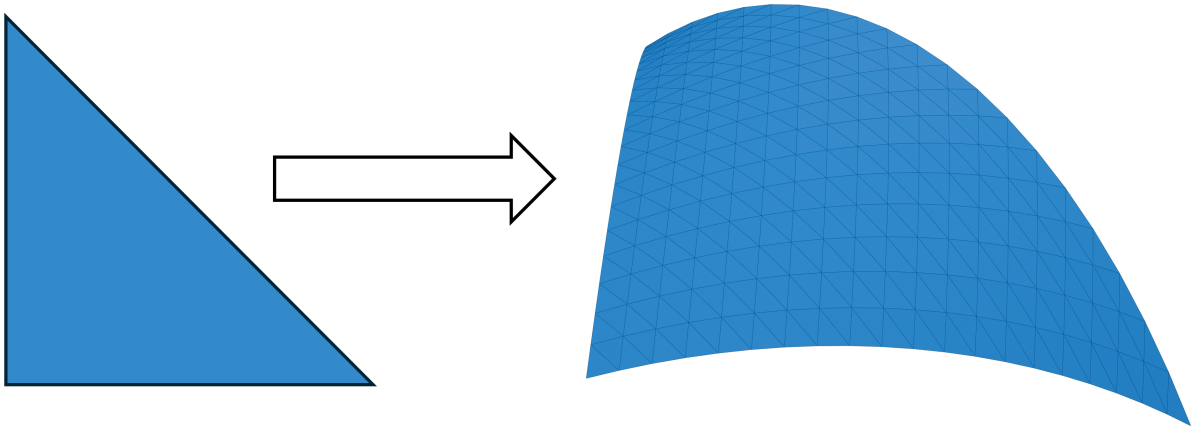


Figure 2.2: Converting a flat triangle into a (cubic) Bézier triangle yields more curvature.

A similar concept is used in Phong Tessellation [1]. Phong Tessellation is a real-time technique designed to make coarse triangle meshes appear smoother without introducing the heavy cost of subdivision or higher-order surfaces. Instead of only interpolated normals for shading, as in Phong shading, it directly adjusts vertex positions during tessellation so that each triangle bends into a curved patch. This produces smoother silhouettes and contours, reducing the blocky appearance of low-polygon meshes while remaining efficient enough for interactive applications.

Displaced subdivision surfaces [14] can act as the predecessor of micro-meshes [21]. Both approaches share the common principle of first constructing a coarse mesh, which is then displaced to represent higher surface details more accurately. In a displaced subdivision surface, a coarse control mesh defines the base structure, which is refined into a smooth domain surface through subdivision. Fine details are then introduced by displacing the smooth surface using a piecewise regular mesh of scalar displacement coefficients.

Micro-meshes (μ-meshes) [21] are an advanced representation technique designed to efficiently encode high-frequency geometric details while maintaining a compact, coarse base structure. A μ-mesh consists of a coarse triangular base mesh coupled with a set of scalar displacement values that define the surface details, as is illustrated in Figure 2.3.

The μ-mesh framework operates by constructing a base mesh from an input surface, which is then subdivided into micro-triangles. Each base vertex is assigned a direction vector that dictates the displacement direction. The displacement values for micro-vertices are determined through ray-casting along the interpolated direction vectors, ensuring that the reconstructed surface closely follows the original high-resolution model.

Subsequent research focused on applying Linear Blend Skinning (LBS) [17] to animate μ-meshes. Since micro-vertices are not explicitly stored in the base mesh, they lack predefined bone weights and matrices. To address this, the barycentric coordinates of a given micro-vertex can be used, along with the skin matrix of each of the corresponding base vertices, to compute a barycentrically interpolated skinning matrix, which is then applied to deform the micro-vertices [8].

2.2. Ray Tracing Displaced Meshes

Ray tracing has become a widely adopted technique for generating photorealistic images. Although traditionally considered computationally expensive and therefore limited to offline rendering, recent advances in hardware – particularly the introduction of dedicated ray tracing

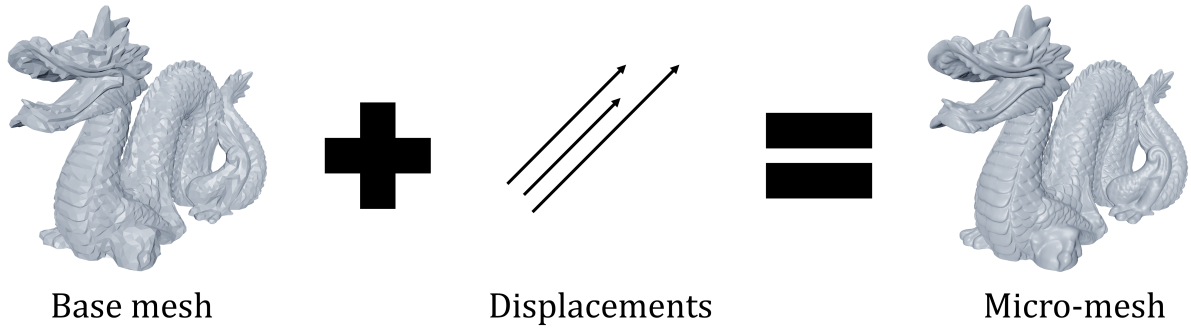


Figure 2.3: The structure of a micro-mesh. A coarse base mesh is displaced according to a set of displacement values to produce the final displaced micro-mesh.

cores in modern GPU's [22] – have significantly accelerated ray-triangle intersection tests and traversal of acceleration structures [24].

μ-meshes are also very efficient for ray tracing. They inherently have a hierarchical structure in which the micro-triangles only need to be checked for intersection when the base triangle is hit. Thonat et al. [29] propose a method for rendering high-quality displacement-mapped surfaces in ray tracing without relying on tessellation. Instead, they construct a displacement acceleration structure, called the Displacement Bounding Volume Hierarchy (D-BVH), which decouples the displacement map from the base mesh. This structure is built directly in texture space using minmax mipmaps [25, 5, 2] and traversed per-ray.

An earlier line of research addressed the challenge of directly ray tracing displacement-mapped geometry without relying on explicit tessellation or excessive memory overhead. Smits et al. [27] introduced an algorithm that operates on triangle meshes with vertex normals and performs ray traversal through displaced micro-triangles generated in barycentric space. Their method significantly reduces storage requirements compared to explicitly generating all displaced geometry, enabling the rendering of highly detailed surfaces with global illumination while maintaining a practical memory footprint.

Lier et al. [15] present a high-resolution compression scheme for ray tracing subdivision surfaces with displacements. Their method employs a two-tier BVH with quantized and compressed leaf representations that maintain geometric fidelity while significantly reducing memory usage. Compared to previous voxel-based approximations, their approach achieves up to a $5:1$ compression ratio with noticeably improved image quality, particularly for close-up views and strongly displaced surfaces.

2.3. Mega Geometry

As of February 2025, NVIDIA's micro-meshes have been deprecated in favor of RTX Mega Geometry [7]. RTX Mega Geometry introduces a cluster-based representation of geometry that enables more efficient construction of acceleration structures and supports massive amounts of animated geometry [12]. By grouping triangles into clusters, the approach reduces build times for bottom- and top-level acceleration structures while maintaining high rendering quality. In addition, Mega Geometry provides continuous level-of-detail mechanisms, allowing geometric detail to be adaptively streamed and refined at runtime.

A path tracer based on Mega Geometry has been implemented by Kraemer et al. [11]. Their work demonstrates the practical use of the framework in a rendering context.

3

Ray Tracing Micro-Meshes

In order to ray trace micro-meshes, we employ a two-dimensional approach and incorporate height field ray tracing to accelerate the process. Let $v_0^B, v_1^B, v_2^B \in \mathbb{R}^3$ denote the vertices of a base triangle. The ray–micro-mesh intersection is defined independently for each base triangle, and since the procedure is identical across all base triangles, the method will be described in detail for a single representative triangle in the following sections.

3.1. Conversion to 2D space

To formulate the problem in 2D, we begin by defining a plane onto which all relevant geometry will be projected and where all computations will be performed. The ray and the displaced triangle are both projected onto this plane, enabling the intersection problem to be addressed entirely in 2D.

3.1.1. Plane creation

The plane is defined by a tangent (T) and a bitangent (B) vector. We will also use the normal (N), which is simply the vector perpendicular to both. The tangent vector is computed as the vector from v_0^B to v_1^B, and the bitangent vector is perpendicular to that:

$$N = (v_1^B - v_0^B) \times (v_2^B - v_0^B)$$

$$T = v_1^B - v_0^B$$

$$B = N \times T$$

where $\times$ is the cross product between two vectors. The origin of the plane is taken to be at v_0^B, meaning all positions on the plane will be expressed relative to this point. After normalizing, T, B, and N form an orthonormal basis for 3D space, which allows us to project back to 3D when we need. The resulting plane, along with the vectors, is illustrated in Figure 3.1.

3.1.2. Ray projection

The ray is projected orthogonally onto the plane defined by the base triangle. Let the ray be given by the parametric equation $r(t) = \mathbf{o} + t\mathbf{d}$, where $\mathbf{o} \in \mathbb{R}^3$ is the ray origin, and $\mathbf{d} \in \mathbb{R}^3$ is the direction. We begin by subtracting the components of $\mathbf{o}$ and $\mathbf{d}$ along N to obtain the projected ray:

$$\mathbf{o}_{\text{proj}} = \mathbf{o} - ((\mathbf{o} - v_0^B) \cdot N) \cdot N$$

$$\mathbf{d}_{\text{proj}} = \mathbf{d} - (\mathbf{d} \cdot N) \cdot N$$

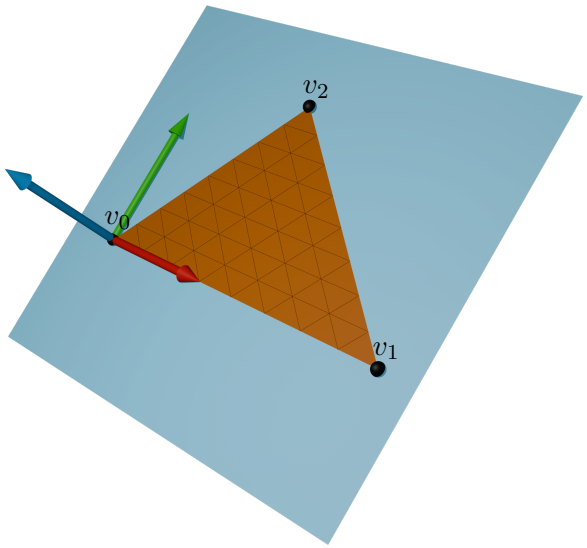


Figure 3.1: Plane that spans the base triangle, along with the tangent (red), bitangent (green) and normal (blue) vectors.

This results in a ray lying entirely within the triangle's plane. To represent it in 2D, we express both $\mathbf{o}_{\text{proj}}$ and $\mathbf{d}_{\text{proj}}$ in the local tangent-bitangent basis of the triangle. The projected ray origin and direction in 2D are computed as:

$$\mathbf{r}_{\text{o}}^{\text{2D}} = \begin{bmatrix} (\mathbf{o}_{\text{proj}} - v_0^B) \cdot T \\ (\mathbf{o}_{\text{proj}} - v_0^B) \cdot B \end{bmatrix}, \quad \mathbf{r}_{\text{d}}^{\text{2D}} = \begin{bmatrix} \mathbf{d}_{\text{proj}} \cdot T \\ \mathbf{d}_{\text{proj}} \cdot B \end{bmatrix}$$

Because the projection is orthogonal, the offset between any point on the original ray and its projection lies along N. Consequently, we can compute the height of the original ray relative to the triangle's plane by taking a dot product with N. This concept is what allows us to do height field ray tracing later on.

3.1.3. Micro-Triangle Displacement within the Plane

During traversal, the micro-triangles are displaced within the base triangle's plane to ensure their position remains consistent with the projection direction N. This is achieved by first displacing each micro-vertex and then orthogonally projecting the resulting positions onto the plane. This is illustrated in Figure 3.2.

We can now introduce some notation. We will use $\triangle_B^{\text{3D}}$ to denote the base triangle in 3D. We will use $\vec{\triangle}^{\text{2D/3D}}$ to denote the (projected) displaced triangle. We will also use $\vec{\triangle}_B^{\text{2D}}$ to specifically talk about the projected displaced base triangle, where only v_0^B, v_1^B, and v_2^B are displaced and orthogonally projected back onto the plane. Where applicable, we will prepend $\triangle$ with μ to indicate that we are talking about a micro-triangle, and add subscript i to denote the subdivision level of the micro-triangle.

We will also use r^{3D} to talk about the original ray, and r^{2D} to denote the projected ray in plane coordinates.

3.2. Micro-triangle traversal

The goal is to determine which micro-triangle is intersected by r^{3D}, but to do so in the 2D domain using r^{2D} that traverses several $\mu\vec{\triangle}^{\text{2D}}$'s.

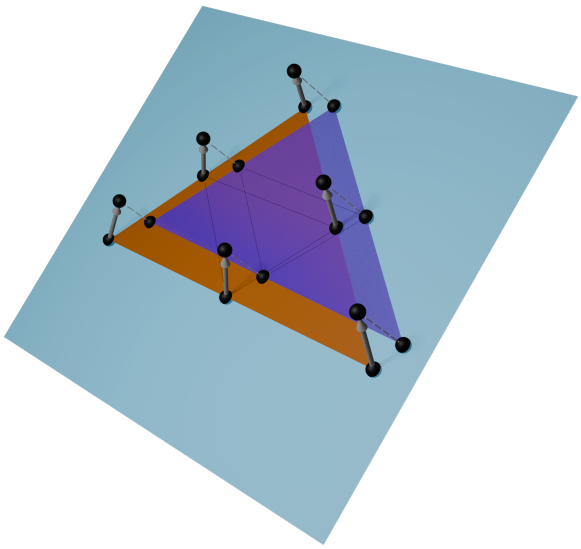
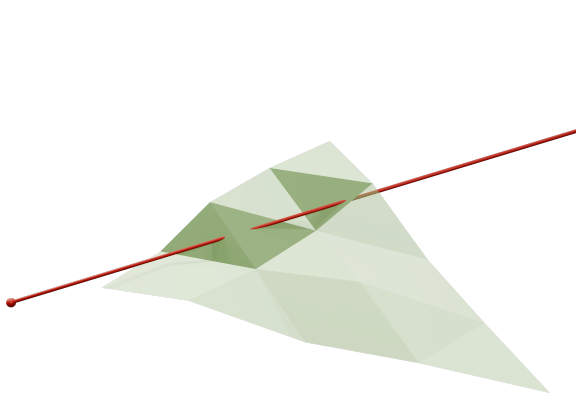


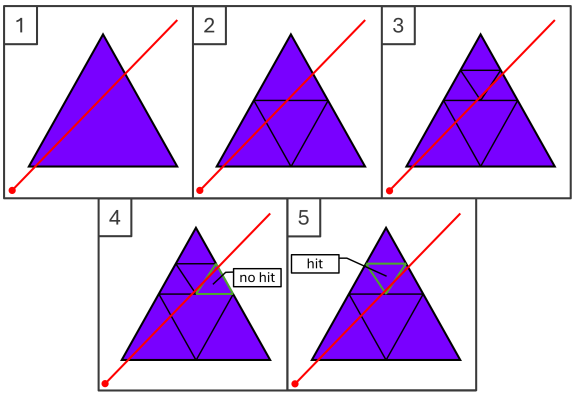
Figure 3.2: Displacing the micro-vertices of the base triangle (orange) and projecting those orthogonally back onto the plane yields a different triangle (purple).

We begin the search for the intersected micro-triangle at subdivision level 0. Note that $\mu\vec{\triangle}_0^{2\mathrm{D}} = \vec{\triangle}_B^{2D}$. At each level i, $\mu\vec{\triangle}_i^{2\mathrm{D}}$ is divided uniformly into 4 smaller micro-triangles $\mu\vec{\triangle}_{i+1}^{2\mathrm{D}}$. Using a depth-first traversal approach, we select one of these sub-triangles and descend into it, proceeding recursively until the finest subdivision level is reached. Only at this level do we consider the actual micro-triangle for intersection.

It is essential that, at each level, we recurse into the micro-triangle that is first traversed by $r^{2\mathrm{D}}$. Arbitrarily selecting a micro-triangle can lead to incorrect results, as illustrated in Figure 3.3. Doing so, may cause the algorithm to miss a micro-triangle that lies earlier along the ray and instead return one that appears later. By always following the first intersection at each level, we guarantee that the final micro-triangle corresponds to the intersected micro-triangle along $r^{3\mathrm{D}}$.

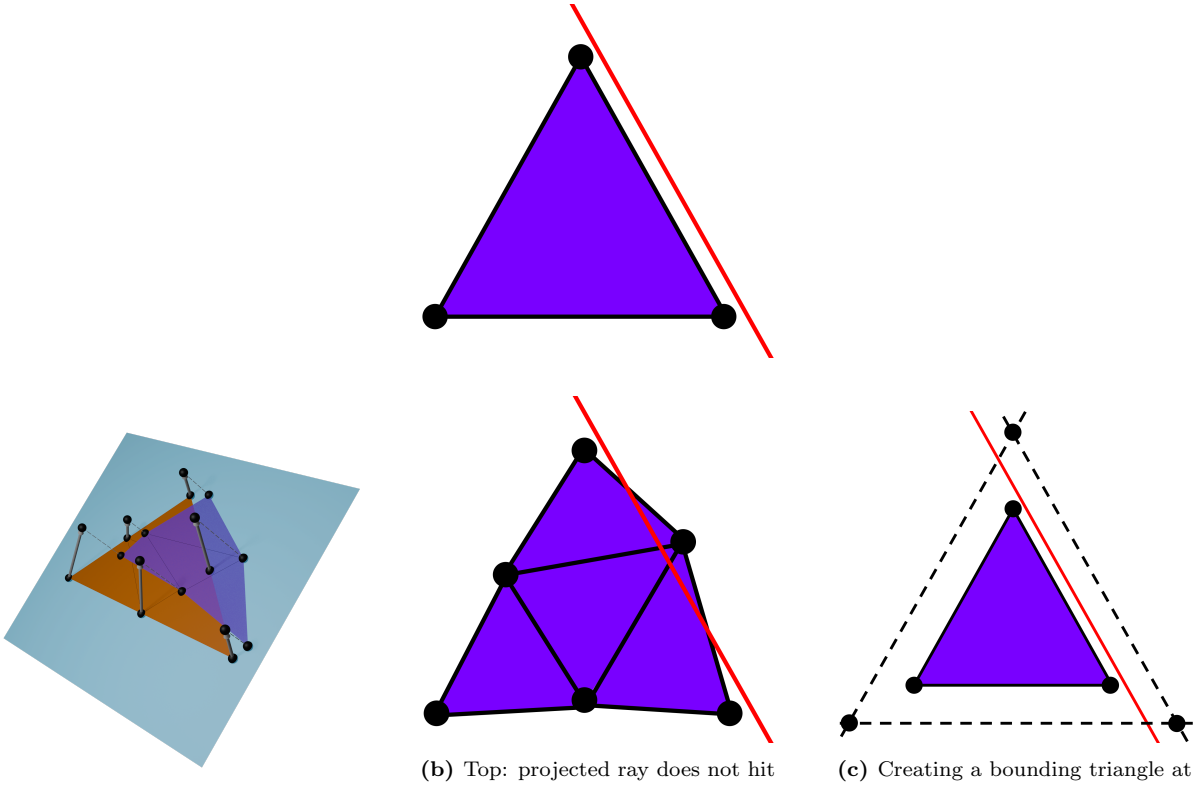


(a) Displaced micro-triangles being hit by a ray. The first micro-triangle being intersected by the ray is the correct `hit`.



(b) If the traversal does not follow the order in which the ray intersects the micro-triangles, an incorrect triangle may be classified as `hit`.

Figure 3.3



(a) Creating the displaced triangle does not always create uniform triangles.

(b) Top: projected ray does not hit triangle at level i. Bottom: projected ray actually hits triangle at level $i+1$.

(c) Creating a bounding triangle at level i that includes all micro-vertices at level $\geq i$ resolves the issue.

Figure 3.4

3.3. Bounding Triangle

Micro-vertices are all displaced in the interpolated direction, but each has a different displacement scale. This means that they are not necessarily uniform triangles, as visible in Figure 3.4a.

A naive method may lead to false negatives during ray-micro-triangle intersection, as illustrated in Figure 3.4b. In the top figure, the ray misses the triangle, so naively checking for intersection against the base triangle will not traverse the hierarchy. However, when we subdivide the triangle, as illustrated in the bottom figure, one can see that the ray actually hits a triangle at a higher subdivision level.

Hence, it is important to conservatively test the top levels to make sure an intersection is not missed in subsequent subdivision levels. A solution for this is to create a *bounding triangle* around the projected base triangle. A bounding triangle is an enlarged version of the given triangle that also includes micro-vertices of subsequent subdivision levels. By including all micro-vertices, we can be sure that r^{2D} does not miss any micro-triangles, since all micro-triangles of future subdivision levels are included in the domain of the bounding triangle. The bounding triangle is visible in Figure 3.4c.

3.3.1. Creating the Bounding Triangle

Several methods can be employed to construct a bounding triangle, as long as it fully encloses all displaced micro-vertices at subsequent subdivision levels. In this work, three different strategies for bounding triangle construction were explored. Each of these approaches is described in the following sections and offers a different trade-off between tightness and memory usage.

Triangle with Minimum Area

The task of constructing a bounding triangle can be formally stated as follows: given a set of n points in the plane, determine a triangle that completely encloses them. Ideally, this triangle should be as small as possible in order to minimize the number of rays that intersect it during ray tracing, thereby improving performance.

The theoretical foundation for this problem was established by Klee and Laskowski [10], who provided a rigorous mathematical characterization of minimal enclosing triangles using concepts from convex geometry and support functions. Building upon this framework, O'Rourke et al. [23] introduced an efficient algorithm with a time complexity of $\Theta(n)$ for computing the minimum-area enclosing triangle of a convex polygon. The method of O'Rourke et al. leverages the rotating calipers technique to systematically examine candidate triangles, enabling practical and effective computation. In applications involving arbitrary point sets, a convex hull is first computed, after which the algorithm is applied to this hull to find the minimum enclosing triangle.

This method requires the storage of six scalar values per bounding triangle. Specifically, the x and y coordinates for each of the triangle's three vertices. While straightforward, this representation becomes impractical for large meshes, as the memory consumption increases rapidly with the number of bounding triangles.

Edge Expansion (EE)

In the edge expansion method, a bounding triangle is constructed by extending each edge of $\mu\vec{\triangle}^{2D}$ outward. For each edge $j \in \{1, 2, 3\}$, we expand the edge outward by a distance δ^j to include the micro-vertex that lies the farthest away from it. This is visualized in Figure 3.5.

By treating each edge independently, this method yields a tight bounding triangle that avoids unnecessary expansion in regions where no micro-vertices of higher subdivision levels extend beyond the original triangle domain. However, this approach necessitates storing a single scalar value per edge, resulting in a total of three scalars per bounding triangle. Due to memory limitations, we adopt a more restrictive method, which is described in the following section.

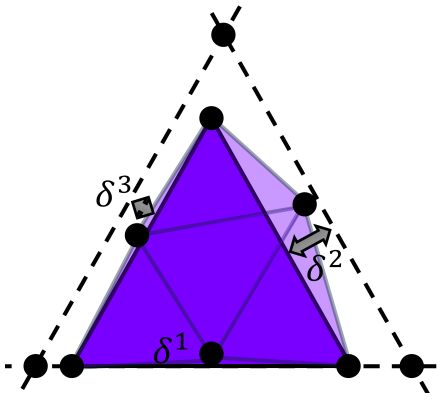


Figure 3.5: A triangle at level i and the micro-triangles at level $i+1$. Edges are displaced by a δ^j to contain all micro-vertices at subsequent subdivision levels.

Uniform Edge Expansion (UEE)

An alternative approach is to expand all three edges uniformly by a single scalar value, defined as $\delta^{\max} = \max(\delta^1, \delta^2, \delta^3)$. This simplification enables the representation of the bounding triangle using only a single scalar value, while still ensuring that all micro-vertices from the current and

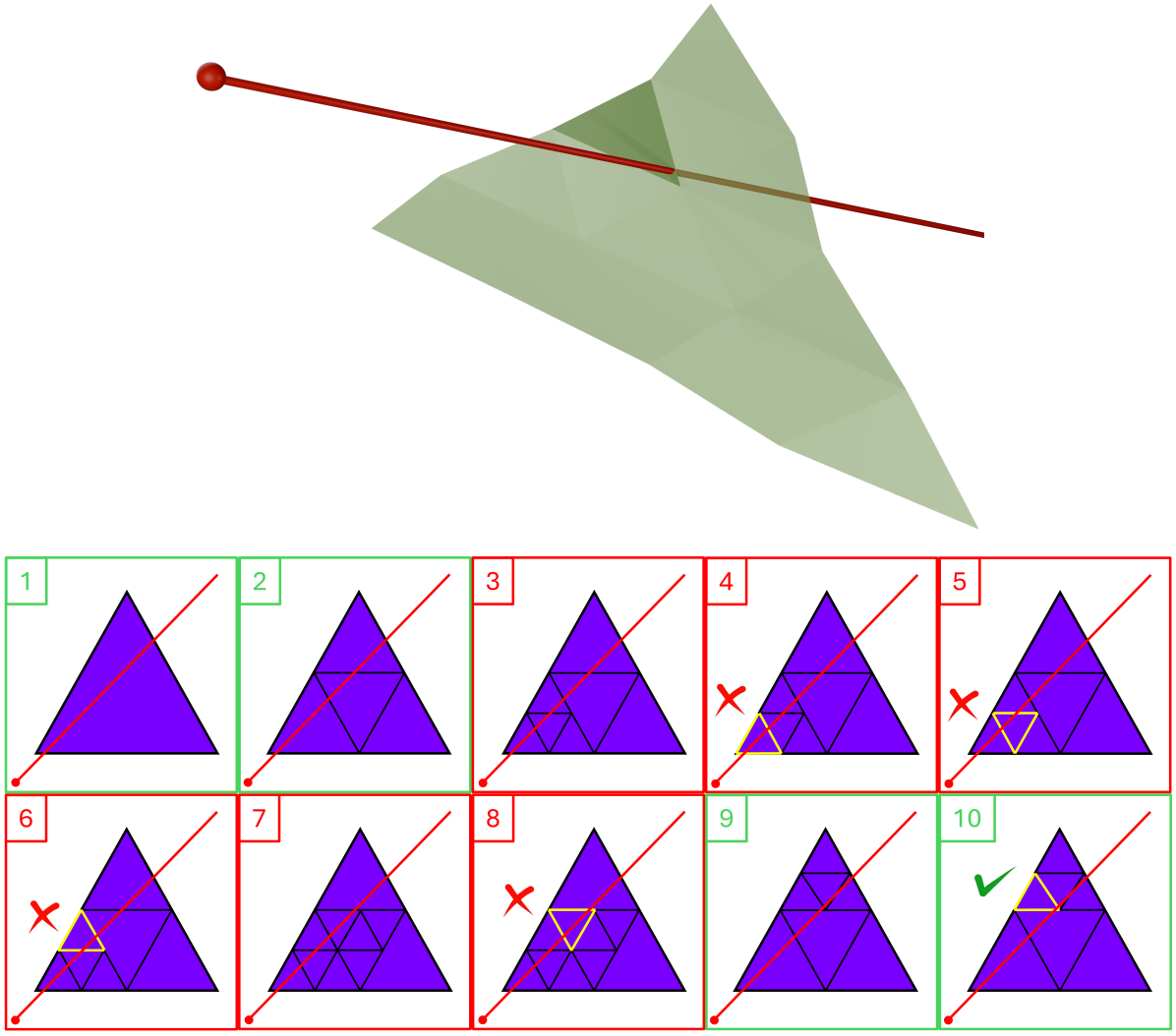


Figure 3.6: Steps of the algorithm for identifying the intersected micro-triangle. The green steps contribute directly to locating the correct intersection, whereas the red steps represent redundant computations.

subsequent subdivision levels are enclosed. Although this strategy does not necessarily yield a minimal bounding triangle, it provides a sufficiently tight approximation for our purposes and offers a favorable trade-off between memory efficiency and spatial coverage.

3.4. Displacement region

We can now fully execute the ray tracing process. As described in Section 3.2, r^{2D} may traverse multiple $\mu\vec{\triangle}^{2D}$, recursively subdividing them in the order of traversal. This process continues until the finest subdivision level is reached, at which point the corresponding micro-triangle is tested for intersection with r^{3D}. However, if the ray passes through several micro-triangles before descending into the correct one, computational resources are wasted on subdivisions that do not contribute to the final intersection result (see Figure 3.6).

To mitigate this inefficiency, we introduce the concept of the *displacement region* as a heuristic to guide the subdivision (see Figure 3.7). The displacement region of a micro-triangle is defined by the minimum and maximum displacement values required to reconstruct $\mu\vec{\triangle}^{3D}$ from its 2D domain representation, by displacing along N.

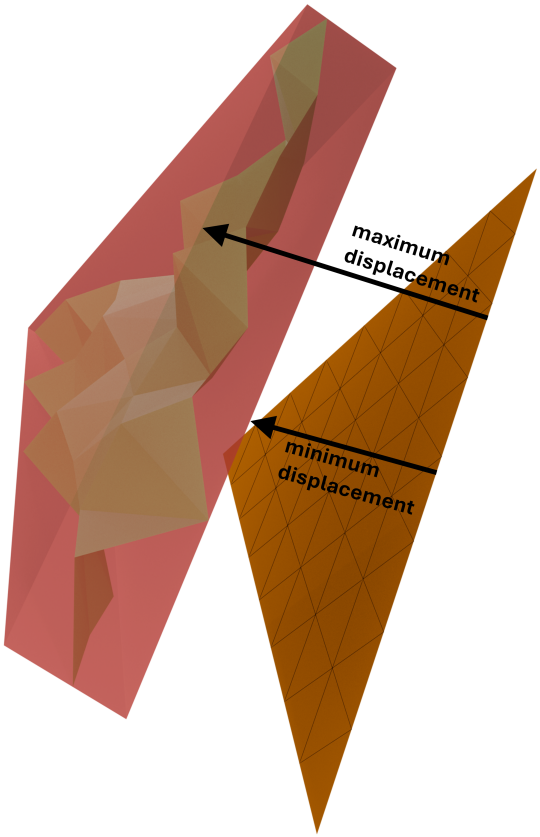


Figure 3.7: The displacement region of a base triangle (orange), shown in red. Displacing the micro-triangles from the base triangle yields the micro-triangles shown in green.

Let t_{entry} and t_{exit} denote the points at which r^{2D} enters and exits a given micro-triangle, respectively. Define $h(t)$ as the distance from a point on t on r^{2D} to r^{3D}, measured along N. If both $h(r^{2D}(t_{\text{entry}}))$ and $h(r^{2D}(t_{\text{exit}}))$ lie strictly outside the bounds of the displacement region—either both below the minimum or both above the maximum displacement—we can safely conclude that r^{3D} cannot intersect any micro-triangle within the current $\mu\vec{\triangle}^{2D}$ (see Figure 3.8). This allows us to terminate recursion early and avoid unnecessary subdivisions.

By using the displacement region, the traversal process is significantly accelerated. In the best case, only a single micro-triangle needs to be visited per subdivision level.

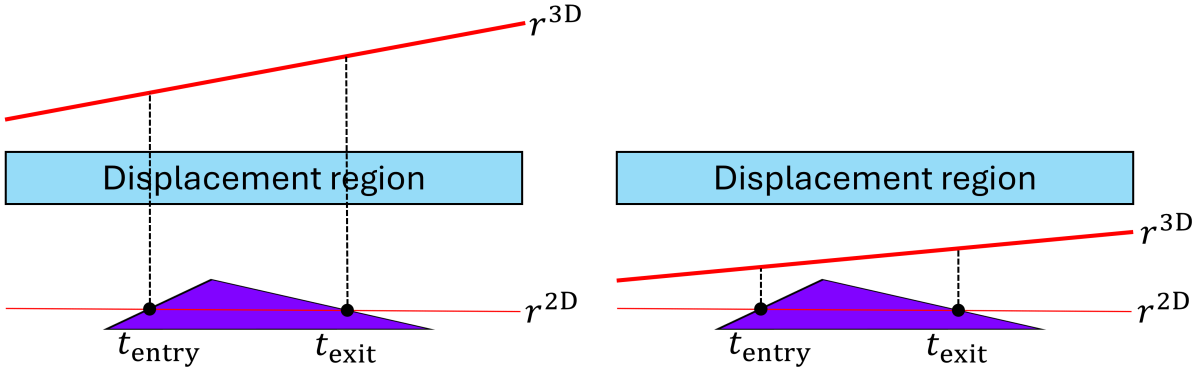


Figure 3.8: Left: t_{entry} and t_{exit} are above the displacement region. Right: t_{entry} and t_{exit} are below the displacement region.

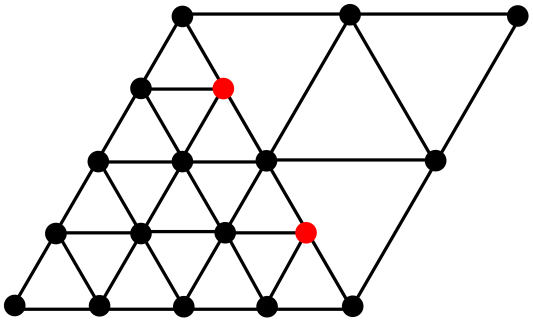


Figure 3.9: Two adjacent (base) triangles, with subdivision level 2 on the left and subdivision level 1 on the right. The red micro-vertices highlight the misalignment on the shared edge, where the (base) triangles do not connect seamlessly.

3.5. Adaptive subdivision level

Up to this point, the process has been described under the assumption that all base triangles share the same subdivision level, which ensures that micro-vertices along shared edges align perfectly. However, when adjacent base triangles differ in their subdivision level, this alignment is no longer guaranteed, as illustrated in Figure 3.9.

In such cases, applying the standard subdivision procedure is not possible, since it may generate micro-vertices along edges where no corresponding vertices exist in the neighbouring triangle. Micro-meshes generated using the method from Maggiordomo et al. [16], however, exhibit the property that adjacent base triangles differ by at most one subdivision level. Specifically, if base triangle A has subdivision level i and a neighbouring base triangle B has subdivision level $i+1$, then the micro-triangles of both meshes are identical for all levels $\leq i$. (If the roles of A and B are reversed, the reasoning is symmetric, and we may always describe the pair as having levels i and $i+1$). Consequently, potential misalignments occur only at the finest subdivision level $i+1$. Only at this level, one must decide whether or not to generate a micro-vertex along a shared edge so that the subdivision remains consistent with the neighbouring base triangle.

This issue, however, is less problematic than it may initially appear. When a neighbouring triangle has a lower subdivision level, micro-triangles lying along the shared edge simply omit the additional micro-vertices. This approach is consistent with the fully tessellated representation provided by Maggiordomo et al. [16], which also supplements the micro-mesh with a complete tessellation of the mesh.

As a result, only six distinct cases can occur, as depicted in Figure 3.10. Three of these involve the omission of two micro-vertices, while the remaining three involve the omission of a single micro-vertex. A configuration in which all three micro-vertices are absent is not possible, since at least one vertex comes from level $\leq i$. It should also be noted that in the latter three cases (one missing micro-vertex), the construction of the micro-triangles may also be performed in an alternative orientation. A complete example illustrating how base triangles with different subdivision levels connect is provided in Figure 3.11.

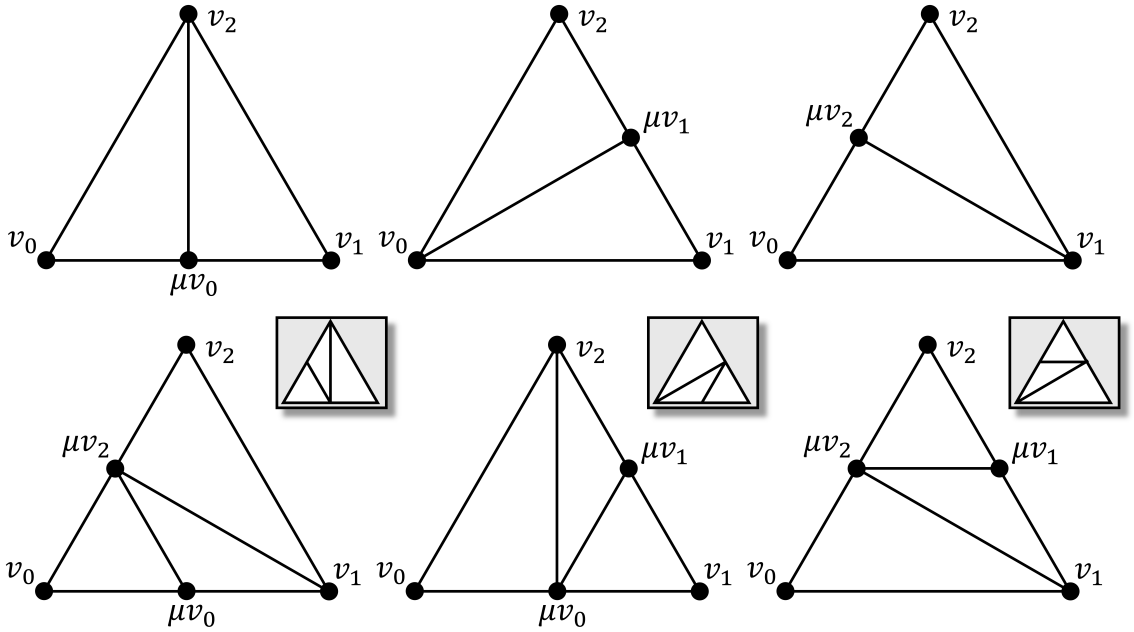


Figure 3.10: Micro-triangles can be constructed accordingly in six different cases. The last three cases have an analogous counterpart, which can equally be used.

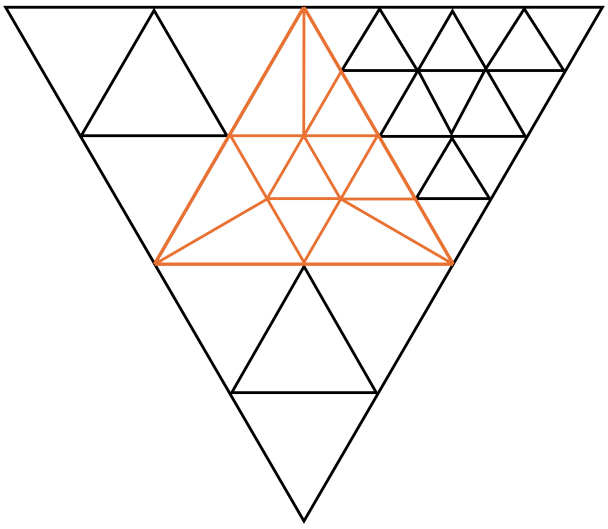


Figure 3.11: A triangle (orange) that is surrounded by other triangles of different subdivision levels.

4

Implementation

The implementation can be divided into two stages: an initialization stage, executed once during setup, and a per-frame execution stage. During initialization, the primary task is to create buffers required for the ray tracer to function. In the per-frame stage, the ray-tracing pipeline itself is executed to compute the final results. Note that no (re)computation of buffers is required in the per-frame stage.

4.1. Buffer Organization

A total of five buffers are required for the ray tracer to operate. The first buffer stores the base vertices of the input mesh, which provide the basis for interpolating per-micro-vertex attributes, such as the displacement direction. The second buffer contains the displacement scales. In the micro-meshes created by Maggiordomo et al. [16], the displacement direction of a micro-vertex can be derived from the base vertices via barycentric interpolation. However, the displacement scale (how far to displace along the direction) cannot be interpolated in the same manner and must therefore be explicitly stored for each micro-vertex in a separate buffer.

The third buffer stores metadata associated with each base triangle, such as indices referencing other buffers. The remaining two buffers are used to store metadata for micro-triangles: the fourth buffer contains the minimum and maximum displacement values, while the fifth buffer stores the edge expansion offsets. Both of these are organized hierarchically: for each subdivision level, data is stored for every micro-triangle, continuing down to the finest subdivision level.

This hierarchical data is arranged in a breadth-first order, such that all the micro-triangle's data at subdivision level i is stored prior to those at level $i + 1$. Since each subdivision step produces four new micro-triangles, the resulting structure corresponds to an implicit quadtree. As in Gruen et al. [8], traversal through this quadtree enables efficient access to the data associated with a given micro-triangle.

As for the second buffer, we similarly require a systematic method to retrieve the displacement scale associated with any given micro-vertex. To this end, we employ a grid-based indexing scheme in which each micro-vertex is assigned an integer coordinate (x, y), with x denoting the row and y denoting the column. An example of such a grid for a level 2 subdivided triangle is illustrated in Figure 4.1. The coordinates of v_0^B, v_1^B, and v_2^B are trivially known. At each subdivision step, a new micro-vertex is created on every edge, and its coordinates are determined as the midpoint of the coordinates of its two adjacent vertices. In other words: given an edge with two endpoints A and B with coordinates (x_A, y_A) and (x_B, y_B) respectively,

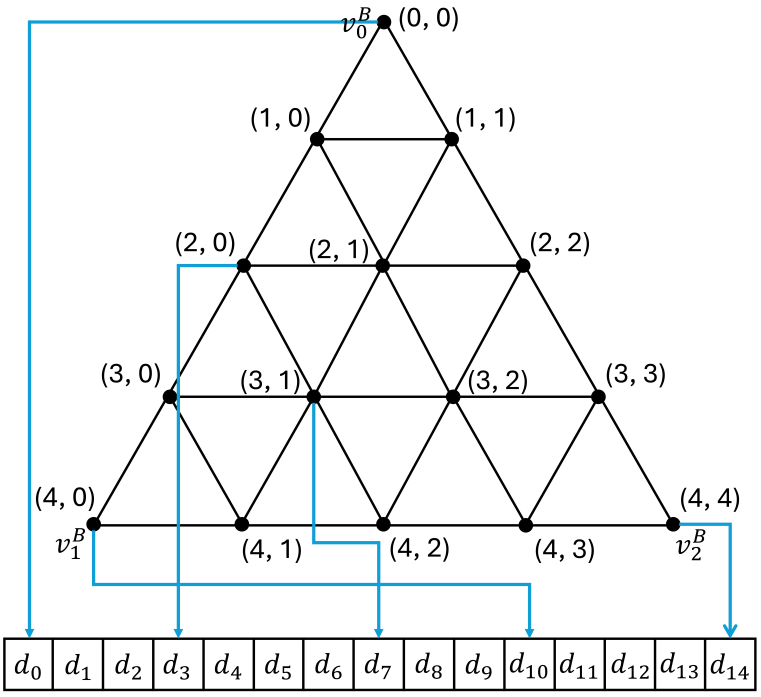


Figure 4.1: Each micro-vertex has an integer coordinate, which is used to compute the index into the buffer that stores the displacement scales. To depict the ordering in which the displacement scales are stored, some arrows are drawn from a micro-vertex to its corresponding displacement scale in the buffer.

the coordinates of the midpoint are calculated as $(\frac{x_A+x_B}{2}, \frac{y_A+y_B}{2})$. A coordinate (x, y) is then converted into a linear index j by summing all numbers until x starting from 0, and adding y to that:

$$j = (\sum_{k=0}^{x} k) + y = \frac{x(x+1)}{2} + y.$$

This specifies the sequential position of the micro-vertex, starting from v_0^B until v_2^B. This index corresponds directly to the storage layout of the displacement scales within the buffer.

4.2. The Ray Tracing Pipeline

Ray tracing micro-meshes integrates naturally into the ray tracing pipeline (see Figure 4.2). After rays are generated and dispatched into the scene, the acceleration structure (AS) is traversed to determine potential intersections.

The acceleration structure consists of two hierarchical levels: the top-level acceleration structure (TLAS) and the bottom-level acceleration structure (BLAS). Conceptually, the TLAS contains references to multiple BLAS instances, while each BLAS stores the geometry of a particular mesh in the scene, as visualized in Figure 4.3. Each BLAS is represented by an axis-aligned bounding box (AABB) that encloses all primitives it contains. When a ray intersects a BLAS, the intersection shader associated with that geometry is invoked to perform user-defined intersection testing and reporting.

Similar to Gruen et al. [8], a single BLAS is constructed for each base triangle. While this design imposes a higher computational cost during acceleration structure construction, it eliminates the need to traverse multiple base triangles in the intersection shader to determine which base triangle was intersected. The any-hit shader is not used in our approach.

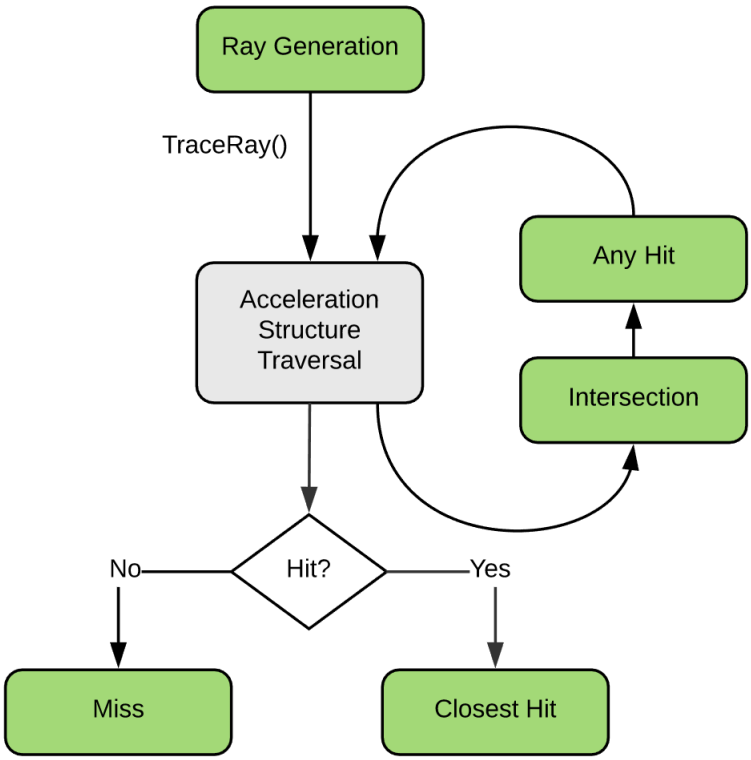


Figure 4.2: The ray tracing pipeline [28].

Because NVIDIA's hardware support for micro-meshes is only introduced starting with the GeForce RTX 40 series [21], the proposed method is implemented entirely in software, primarily within the intersection shader. Once a valid intersection is reported, the closest-hit shader is executed to compute the final shading. If no intersection is found, the miss shader is invoked, which typically returns a background color.

4.3. Pseudocode

The core functionality of the proposed ray tracer is implemented in the intersection shader. Pseudocode for this shader is provided in Algorithm 2, and the structures it relies on are defined in Algorithm 1. The pseudocode serves as a high-level overview of the algorithmic steps, abstracting away implementation-specific details for clarity.

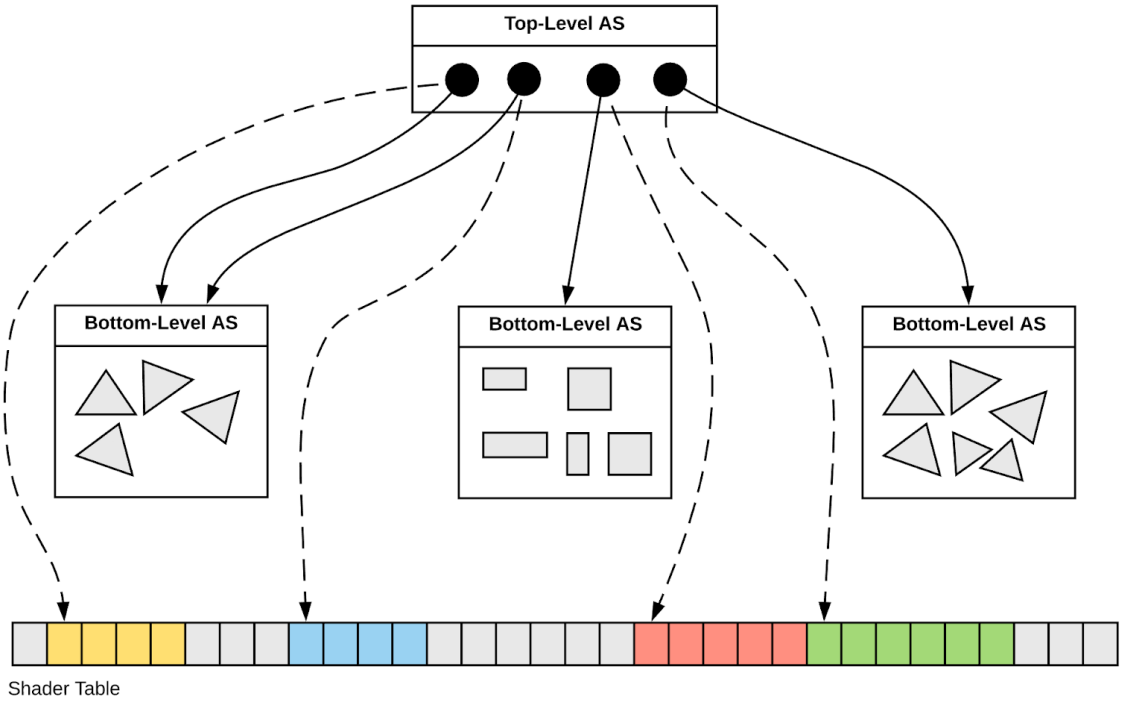


Figure 4.3: The acceleration structure used in the ray tracing pipeline [28]. The shader table can be ignored.

Algorithm 1 Structured used in Algorithm 2

```
structure BaseVertex
    position
    direction
    ▷ More fields can be added ◁
structure Plane
    T, B, N                                   ▷ Tangent, bitangent, normal
    origin
    methods
        convertTo3D(p)      ▷ Converts a 2D point p on the plane to 3D coordinates
        convertTo2D(p)      ▷ Converts a 3D point p to 2D plane coordinates
structure Vertex2D
    position
structure Triangle2D
    vertices[3]
    subDivLvl
structure Ray2D
    origin
    direction
structure Edge
    start, end
    methods
        middle()            ▷ Creates and returns a vertex at the middle of the edge
```

Algorithm 2 Ray tracing micro-meshes in 2D

```
1: function isInsideTriangleDomain(Vertex2D vertices[3], Ray2D r, minDispl, maxDispl)
2:     return isWithinTriangleBounds(ray, vertices) ∧
         isInsideDisplacementRegion(ray, minDispl, maxDispl)
3: function subdivideTriangle(Triangle2D t, Ray2D r)
4:     Vertex2D v_0, v_1, v_2 ← t.vertices
5:     Vertex2D μv_0 ← Edge(v_0, v_1).middle()
6:     Vertex2D μv_1 ← Edge(v_1, v_2).middle()
7:     Vertex2D μv_2 ← Edge(v_2, v_0).middle()
8:     intersectedTriangles ← []
9:     for all μt ∈ {{v_0, μv_0, μv_2}, {μv_0, v_1, μv_1}, {μv_2, μv_1, v_2}, {μv_0, μv_1, μv_2}} do
10:        v_0^displ, v_1^displ, v_2^displ ← createDisplacedTriangle(μt)
11:        microTriDelta ← deltas[...]                                  ▷ From buffer
12:        v_0^bound, v_1^bound, v_2^bound ← createBoundingTriangle(v_0^displ, v_1^displ, v_2^displ, microTriDelta)
13:        minDispl, maxDispl ← minMaxDisplacements[...]                ▷ From buffer
14:        if isInsideTriangleDomain(v_0^bound, v_1^bound, v_2^bound, r, minDispl, maxDispl) then
15:            tri ← Triangle2D(μt, t.subDivLvl + 1)
16:            intersectedTriangles.add(tri)

17:    sort(intersectedTriangles, r)        ▷ Sort in order of how the ray intersected them
18:    return intersectedTriangles
19: function rayTraceMicroMeshTriangle(Triangle2D t, Ray2D r, Plane p, directions[3])
20:    if t.subDivLvl = MAX_SUB_DIV_LVL then
21:        v_0 ← p.convertTo3D(t.vertices[0].position) +
           computeDisplacement(t.vertices[0], directions)
22:        v_1 ← p.convertTo3D(t.vertices[1].position) +
           computeDisplacement(t.vertices[1], directions)
23:        v_2 ← p.convertTo3D(t.vertices[2].position) +
           computeDisplacement(t.vertices[1], directions)
24:        return rayTraceTriangle(v_0, v_1, v_2)          ▷ Usual ray-triangle intersection
25:    else
26:        for all tri ∈ subdivideTriangle(t, r) do
27:            if rayTraceMicroMeshTriangle(tri, r) then
28:                return

29: function main
30:    BaseVertex v_0^B, v_1^B, v_2^B ← vertices[...]                   ▷ From buffer
31:    Plane plane ← createPlane(...)
32:    ▷ Create the displaced triangle                                  ◁
33:    Vertex2D v_0 ← plane.convertTo2D(v_0^B.position)
34:    Vertex2D v_1 ← plane.convertTo2D(v_1^B.position)
35:    Vertex2D v_2 ← plane.convertTo2D(v_2^B.position)
36:    subDivisionLevel ← 0
37:    Triangle2D triangle ← Triangle2D(v_0, v_1, v_2, subDivisionLevel)
38:    ▷ Project 3D ray to plane                                        ◁
39:    Ray2D r ← projectRay(...)
40:    ▷ Check if projected ray lies outside triangle domain            ◁
41:    v_0^displ, v_1^displ, v_2^displ ← createDisplacedTriangle(triangle)
42:    baseTriDelta ← deltas[...]                                       ▷ From buffer
43:    v_0^bound, v_1^bound, v_2^bound ← createBoundingTriangle(v_0^displ, v_1^displ, v_2^displ, baseTriDelta)
44:    minDispl, maxDispl ← minMaxDisplacements[...]                    ▷ From buffer
45:    if ¬isInsideTriangleDomain(v_0^bound, v_1^bound, v_2^bound, r, minDispl, maxDispl) then
46:        return
47:    rayTraceMicroMeshTriangle(triangle, ray, plane, [v_0^B.direction, v_1^B.direction, v_2^B.direction])
```

5

Results and Discussion

For evaluation, several meshes were tested, a selection of which is shown in Figure 5.1. All experiments were conducted using an application developed with DirectX Raytracing (DXR) [18]. The tests were conducted on an NVIDIA GeForce RTX 3060 Ti with 8GB of memory, and on Windows 11.

5.1. Memory Analysis

For this work, the primary objective was to minimize memory consumption, thereby enabling the rendering of meshes with a high number of (micro-)vertices and (micro-)triangles, while maintaining acceptable runtime performance. Memory footprint was of particular importance, as two buffers store hierarchical data whose size increases rapidly with higher subdivision levels.

5.1.1. Cost Function

To analyze memory usage, we examine the impact of different bounding triangle constructions, as described in Section 3.3.1, for a single base triangle and all its associated micro-triangles. We begin with a basic memory cost function:

$$\text{cost}(L) = 24 + \frac{(2^L + 1)(2^L + 2)}{2} + c \cdot \sum_{i=0}^{L} 4^i$$

where L denotes the maximum subdivision level of the base triangle, and the cost function is expressed in terms of 32-bit scalar values.

The cost function consists of three terms. The first term is a constant contribution corresponding to six indices into other buffers, and the storage of position and direction data for the three base vertices: $6 + 3 \cdot 2 \cdot 3 = 24$.

The second term accounts for the number of micro-vertices in the triangle, and each micro-vertex requires storage of a single displacement scale. The third term reflects the amount of hierarchical data stored. Since each subdivision level introduces four new micro-triangles per micro-triangle, this is expressed as a summation over 4^i.

5.1.2. Influence of Different Bounding Triangles on Memory

Different bounding triangle construction methods influence the constant c in the hierarchical storage cost. Using a minimum-area bounding triangle requires storing the 2D coordinates

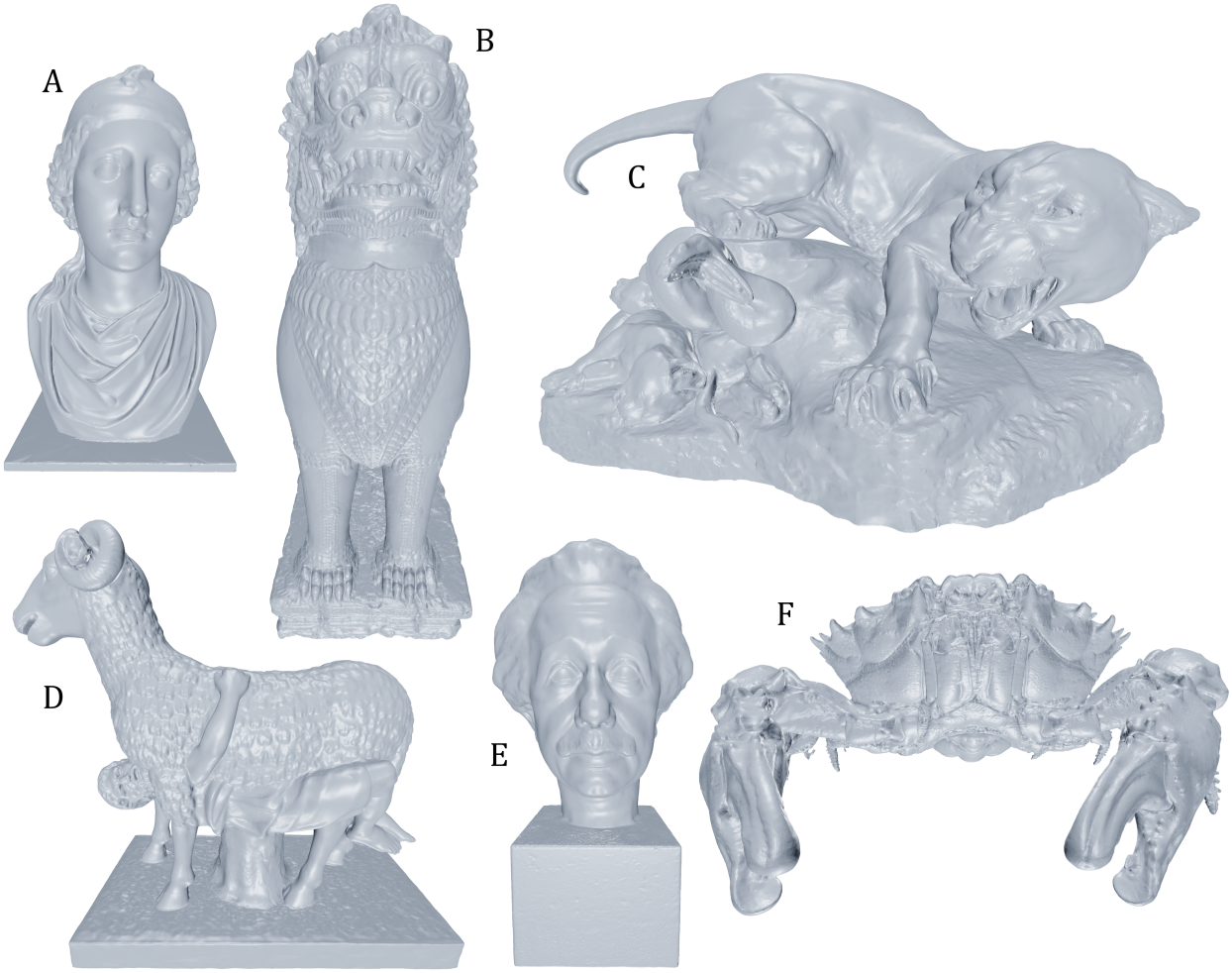


Figure 5.1: A selection of micro-meshes ray-traced by the developed method.

of its three vertices (six scalar values), together with two scalar values for the minimum and maximum displacement, yielding $c = 8$. With similar reasoning, the edge-expansion method results in $c = 5$, while uniform edge expansion reduces this further to $c = 3$. The effect of these alternatives on overall memory consumption is reported in Table 5.1.

Level	Minimum area	EE	UEE
0	35	32	30
1	70	55	45
2	207	144	102
3	749	494	324
4	2905	1882	1200
5	11505	7410	4680

Table 5.1: Comparison of memory consumption for different bounding-triangle construction methods: minimum area, edge expansion, and uniform edge expansion. All reported values correspond to counts of 32-bit scalars.

Finally, it is worth noting that memory usage could be further reduced by omitting the storage of minimum and maximum displacements, i.e., not employing height field ray tracing. However, this optimization was found to incur a big runtime penalty due to the inability to discard non-intersecting triangles early.

5.1.3. Comparison

For comparison, the proposed approach is evaluated against traditional ray tracing, where the micro-mesh is fully tessellated to the maximum subdivision level and the resulting mesh is uploaded to the ray tracer as a regular mesh. In the remainder of this thesis, the notation $\mathcal{M}_T$ is used to denote the fully tessellated mesh rendered through traditional ray tracing, while $\mathcal{M}_\mu$ refers to the ray-traced micro-mesh.

The memory cost of this traditional ray tracing is computed analogously to our method, namely by accounting for all data stored per triangle. In this case, the cost reduces to the storage of vertex attributes. For our analysis, we assume that each vertex stores a position and a normal. The results of this memory analysis are reported in Table 5.2.

Level	Cost
0	18
1	36
2	90
3	270
4	918
5	3366

Table 5.2: Memory analysis of traditional ray tracing for a single triangle. All reported values correspond to counts of 32-bit scalars.

5.1.4. Optimized Buffer Storage

When comparing the two tables, it can be observed that the memory consumption of our approach initially exceeds that of traditional ray tracing, which would make the method impractical in its current form. To address this, we introduce an optimization that significantly reduces memory usage: hierarchical data are not stored for the finest subdivision level, but are instead computed on the fly. This is feasible because the hierarchical data consists of only two components: the bounding triangle and the minimum and maximum displacement. At the finest level, the bounding triangle coincides with the micro-triangle itself, while the minimum and maximum displacements can be directly obtained from the three displacement scales of its vertices. This strategy yields substantial memory savings, as illustrated in Table 5.3, since higher subdivision levels introduce an exponential increasing number of micro-triangles. Consequently, omitting the finest level provides the greatest relative reduction in memory requirements.

Level	UEE	Traditional	Difference
0	27	18	+9 (+50%)
1	33	36	-3 (-8.33%)
2	54	90	-36 (-40%)
3	132	270	-138 (-51.11%)
4	432	918	-486 (-52.94%)
5	1608	3366	-1758 (-52.23%)

Table 5.3: Memory consumption expressed in the number of 32-bit scalars when excluding storage at the finest subdivision level. For ease of comparison, the corresponding values for traditional ray tracing are also included.

	Level	#Scalars per vertex attribute			
		6	9	12	15
Traditional	0	18	27	36	45
UEE		27	36	45	54
Traditional	1	36	54	72	90
UEE		33	42	51	60
Traditional	2	90	135	180	225
UEE		54	63	72	81
Traditional	3	270	405	540	675
UEE		132	141	150	159
Traditional	4	918	1377	1836	2295
UEE		432	441	450	459
Traditional	5	3366	5049	6732	8415
UEE		1608	1617	1626	1635

Table 5.4: Memory analysis for a singe base triangle for both traditional ray tracing and UEE. All reported values correspond to counts of 32-bit scalars.

Since uniform edge expansion yields the lowest memory consumption, we restrict the comparison to this method and traditional ray tracing. The memory usage of our ray tracer can thus be expressed as:

$$\text{cost}(L) = 24 + \frac{(2^L + 1)(2^L + 2)}{2} + 3 \cdot \sum_{i=0}^{L-1} 4^i$$

5.1.5. Impact of Vertex Attributes on Memory Efficiency

It is important to note that the memory advantage of our approach increases with the amount of interpolatable data stored at the base vertices. In the present analysis, we assumed two vertex attributes. However, if additional interpolatable attributes such as texture coordinates or skinning matrices [8] are included, the relative benefit becomes more significant, as illustrated in Table 5.4 and Figure 5.2.

Figure 5.2 illustrates that the curves corresponding to UEE become progressively flatter as the subdivision level increases, indicating that the relative advantage of UEE grows with higher subdivision levels. Moreover, UEE exhibits a slower growth in memory consumption compared to traditional ray tracing, suggesting that the proposed approach scales more favorably for larger meshes. With the exception of subdivision level 0, UEE consistently outperforms the traditional approach in terms of memory efficiency.

5.2. Level of Detail

NVIDIA's micro-meshes support up to subdivision level five [19]. While it may be tempting to consistently employ subdivision level five, given its capacity to capture fine geometric detail, the visual improvement beyond level three is often marginal (see Figure 5.3).

Naturally, higher subdivision levels incur greater memory costs. A mesh in which all base triangles can be subdivided to the highest possible subdivision level guarantees preservation of fine detail, but it also introduces unnecessary refinement in regions where no such precision is required (see Figure 5.4).

Meshes with adaptive subdivision levels address this issue by assigning higher subdivision levels only to regions containing fine detail, while applying lower levels where little to no

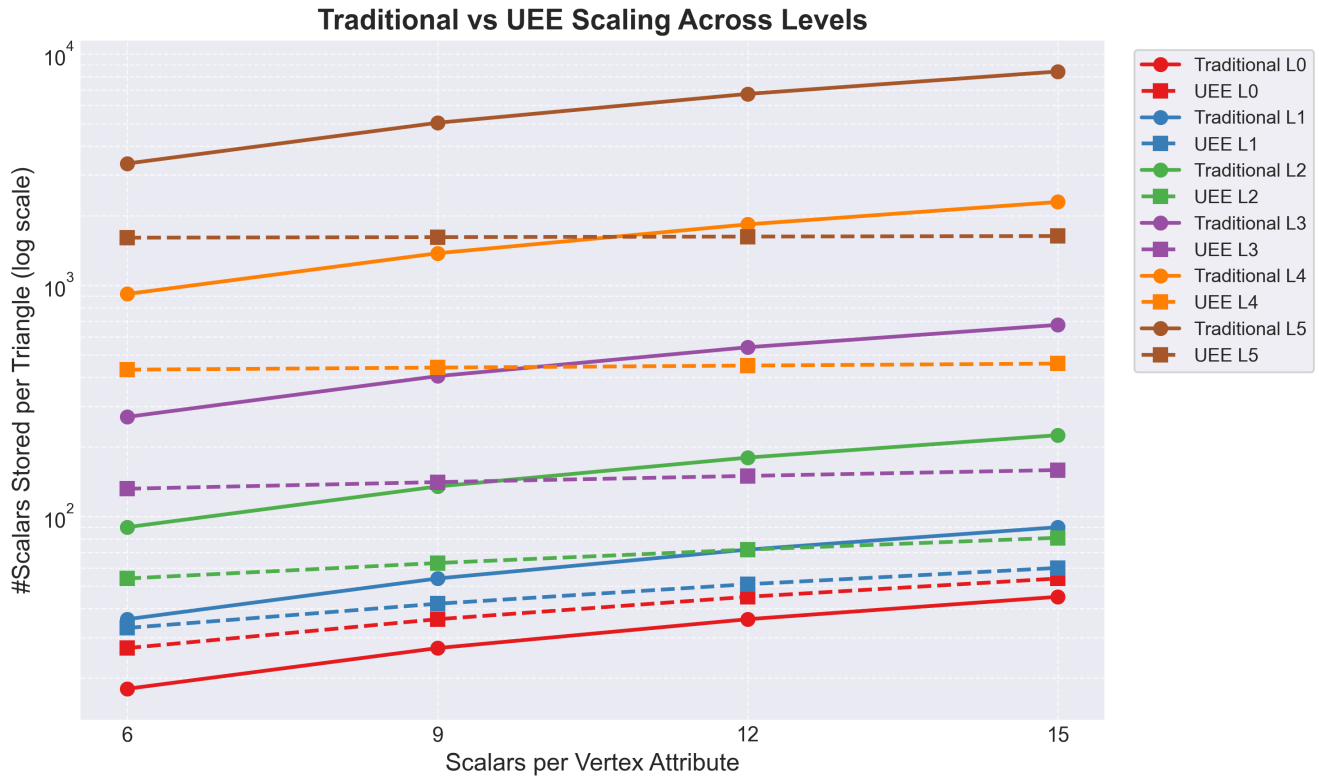


Figure 5.2: Visualization of Table 5.4. The y-axis is logarithmically scaled to accommodate the wide range of values. Except for level 0, UEE outperforms traditional ray tracing in terms of memory.

displacement occurs. As a result, visual fidelity is largely preserved while reducing memory overhead. Figure 5.5 illustrates this effect by comparing a uniform level-5 micro-mesh with a micro-mesh employing varying subdivision levels.

5.3. Inspection of Evaluated Micro-Meshes

To evaluate the ray tracer on practical meshes, we measured both runtime performance and GPU memory consumption for the meshes shown earlier in Figure 5.1. Prior to these experiments, before converting to micro-meshes, the input meshes were preprocessed using MeshLab [3] to reduce their triangle count and thereby improve the quality of the base mesh. The results of this evaluation are summarized in Table 5.5, with all experiments conducted at full HD resolution.

It can be observed that both runtime and memory consumption increase with higher subdivision levels. This behavior is expected, as finer subdivisions require storing a larger number of micro-triangles. In terms of runtime, the primary reason for this increase is the deeper recursion required during intersection testing. Specifically, a subdivision level i corresponds to a recursion depth of i, which must be traversed to determine the intersected micro-triangle. This deeper recursion introduces additional memory accesses, leading to execution stalls that accumulate over time and ultimately constitute the principal performance bottleneck of the developed ray tracer.

Another noteworthy trend is that micro-meshes with varying subdivision levels incur only a fraction of the runtime and memory costs compared to those uniformly subdivided to level 5. In terms of runtime, their performance is comparable to subdivision level 3 or 4, while

Figure 5.3: A micro-mesh ray-traced at subdivision levels 0 through 5, illustrating the progressive refinement of geometric detail.

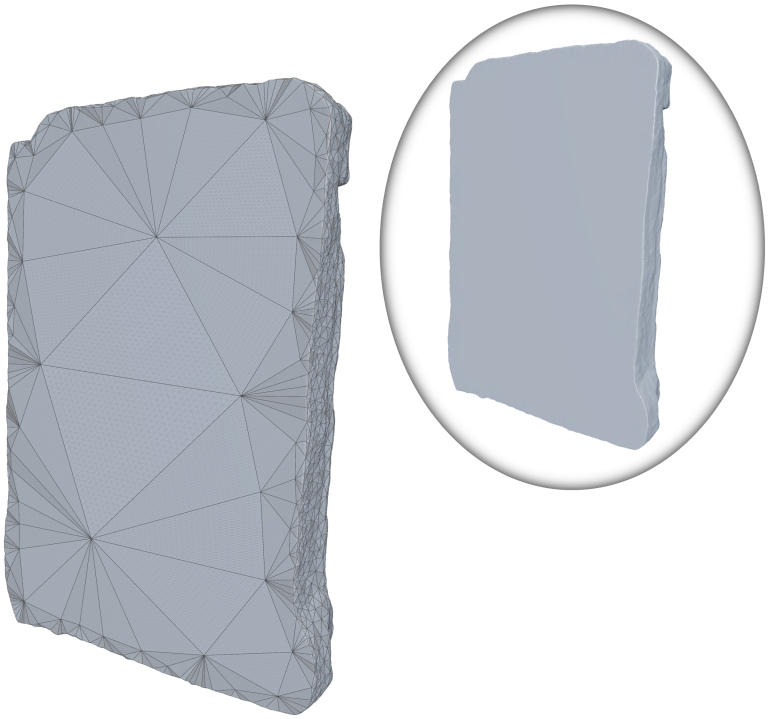


Figure 5.4: Uniform subdivision of a micro-mesh to level 5. The back of the mesh is also subdivided to this level, despite being entirely flat and not requiring such a high (or any) subdivision level.

their memory consumption consistently remains close to that of level 3. Together with the observation in Section 5.2 that geometric detail beyond subdivision level 3 is often visually indistinguishable, this demonstrates that micro-meshes with adaptive subdivision levels achieve a favorable balance between runtime efficiency, memory usage, and visual quality.

For comparative analysis, the meshes shown in Figure 5.1 were also evaluated as $\mathcal{M}_T$, with the corresponding results presented in Table 5.6. The evaluations were performed under the same conditions as for $\mathcal{M}_\mu$. Memory consumption was computed as the combined size of the vertex and index buffers, where each vertex stores both position and normal attributes.

In terms of runtime, all meshes achieved frame times below 6 ms. Traditional ray tracing performs approximately 2–2.5× faster than the proposed method due to the use of hardware-accelerated ray tracing, whereas the proposed method operates entirely in software. However, in terms of memory efficiency, the traditional approach performs significantly worse, requiring approximately 4× more memory than the proposed method.

5.4. Runtime Analysis

The runtime values reported in Table 5.5 and Table 5.6 should be interpreted with caution. Although measurements were conducted as objectively as possible, the frame rate is strongly influenced by the camera position within the scene. When the camera is placed close to the mesh, the mesh occupies a larger portion of the screen, resulting in a greater number of shader invocations. Conversely, when the camera is positioned farther away, the mesh covers a smaller screen area, leading to fewer invocations. This directly affects the measured runtime, as an increased number of shader executions translates to higher per-frame computation time.

Since the evaluated meshes differ in size, it was not feasible to use a single fixed camera position across all test cases. Instead, for each mesh, the camera was positioned at an intermediate

Level 5 Varying levels

Figure 5.5: Quality comparison between a uniformly subdivided micro-mesh at level 5 and a micro-mesh with varying subdivision levels.

$\mathcal{M}_\mu$	Runtime (ms)					Memory (MB)				
Level = ...	2	3	4	5	0–5	2	3	4	5	0–5
Aphrodite (A)	6.37	9.15	11.43	14.90	9.93	1.33	3.95	14.01	53.47	3.50
Bayon Lion (B)	6.40	8.53	11.37	16.76	11.62	2.34	6.92	24.53	93.56	6.68
Lion with Snake (C)	9.82	14.97	19.06	24.36	16.70	2.00	5.92	21.02	80.20	6.31
Ram (D)	7.59	10.36	13.53	17.77	14.37	3.86	11.40	40.39	154.04	11.40
Einstein (E)	5.93	7.86	9.91	13.41	9.96	1.06	3.14	11.13	42.47	3.48
Crab (F)	8.83	12.79	17.25	22.30	17.32	3.21	9.47	33.56	128.00	9.60

Table 5.5: Runtime and memory usage of the ray-traced micro-meshes ($\mathcal{M}_\mu$) for several subdivision levels. The column 0–5 reports values for a micro-mesh with varying subdivision levels.

$\mathcal{M}_T$	Runtime (ms)					Memory (MB)				
Level = ...	2	3	4	5	0–5	2	3	4	5	0–5
Aphrodite (A)	5.92	5.81	5.96	5.97	5.79	4.28	15.12	56.10	215.27	12.07
Bayon Lion (B)	5.88	5.94	5.94	5.91	5.95	7.50	26.52	98.36	377.29	24.64
Lion with Snake (C)	5.89	5.92	5.89	5.92	5.91	6.41	22.71	84.27	323.33	17.89
Ram (D)	5.87	5.95	5.95	5.89	5.91	12.37	43.68	161.97	621.20	42.54
Einstein (E)	5.91	5.91	5.96	5.98	5.96	3.41	12.04	44.66	171.29	12.33
Crab (F)	5.89	5.93	5.93	5.91	5.94	10.28	36.30	134.59	516.20	35.19

Table 5.6: Runtime and memory usage of the fully tessellated meshes ($\mathcal{M}_T$) for several subdivision levels. The column 0–5 reports values for meshes in which, prior to tessellation, base triangles had varying subdivision levels.

distance and remained fixed across all subdivision levels of that mesh. Consequently, the runtime measurements should be analyzed on a per-mesh basis, rather than being directly compared across different meshes.

5.5. Image Difference

Since no approximations are introduced in the proposed method, $\mathcal{M}_\mu$ should, in theory, produce results identical to those obtained from $\mathcal{M}_T$. To verify this, the error between the two approaches was computed by evaluating the pixel-wise image difference. An example of this comparison for the Actaeon model (Figure 5.3, level 1) is shown in Figure 5.6.

Upon close examination, bright pixels are visible, representing an amplified visualization of small non-zero error values. Manual inspection revealed that these deviations primarily occur along edges, likely resulting from floating-point precision errors during computation. Nevertheless, these discrepancies are minimal and imperceptible to the human eye, as can be inspected in Figure 5.7.

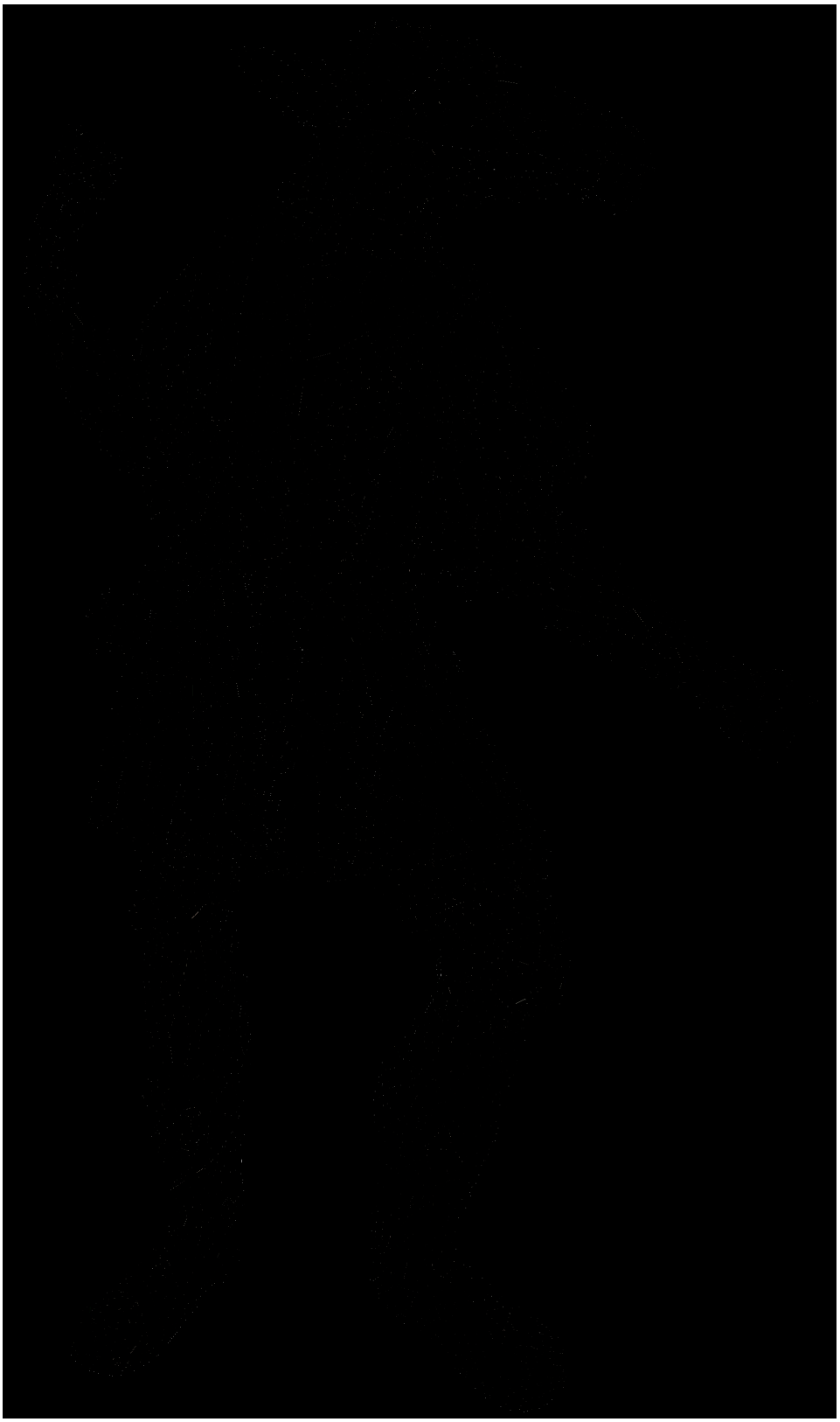


Figure 5.6: Image difference of Actaeon (Figure 5.3, level 1). The brightness has been amplified by a factor of 10 to enhance the visibility of the errors, which are otherwise too subtle to be perceived.

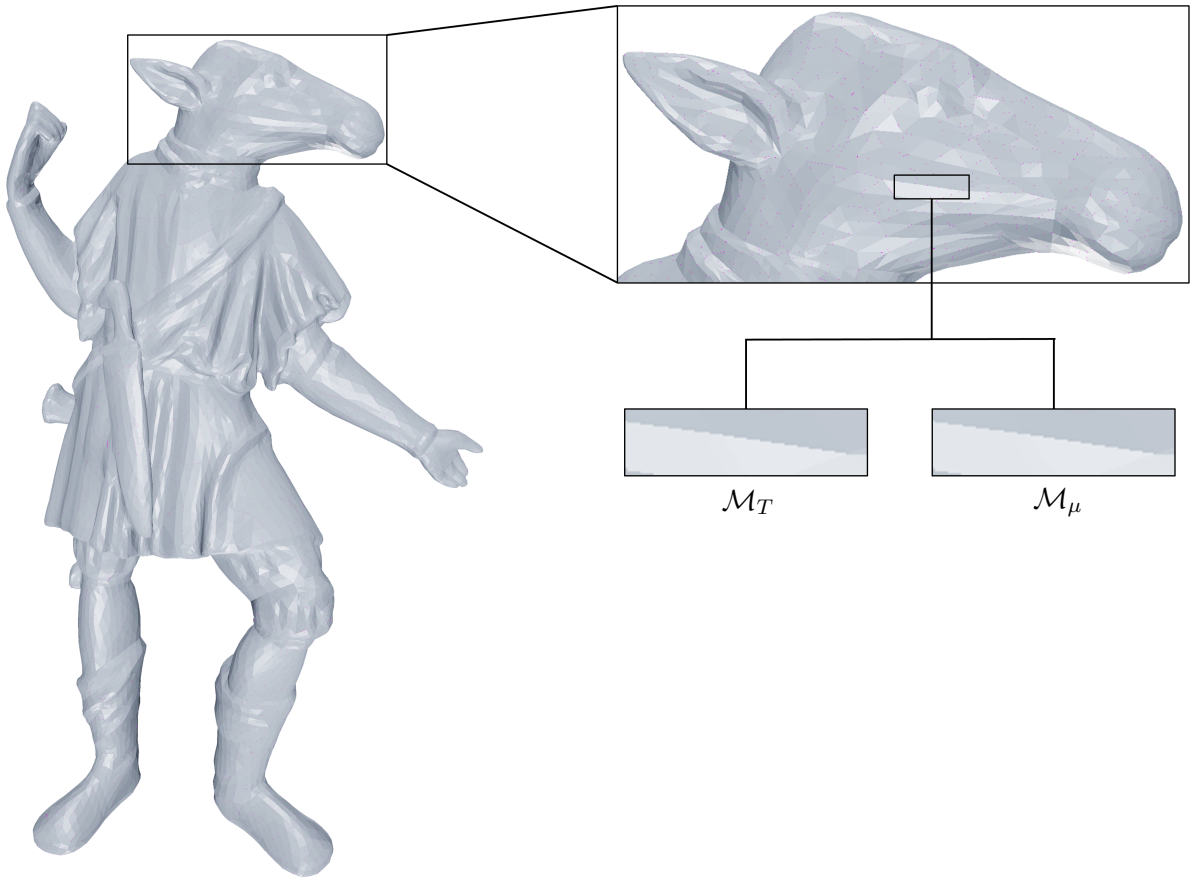


Figure 5.7: Error highlighting of Figure 5.6. Every purple pixel is a non-zero error. The visual impact, however, is negligible.

6

Conclusion

This thesis has presented the research, development, and implementation of a ray tracer capable of rendering NVIDIA's micro-meshes created using the method by Maggiordomo et al. [16].

The ray is orthogonally projected onto the plane spanned by the base triangle. A hierarchical traversal of the micro-triangles is then performed, starting at level 0 and proceeding to the finest subdivision level, with subdivision occurring only where necessary. To prevent false negatives during traversal, a bounding triangle is employed, while a displacement region guides the selection of micro-triangles to be subdivided. Once the finest subdivision level is reached, the corresponding micro-triangle is displaced into 3D space, where a ray-triangle intersection test is performed.

The proposed approach offers two main advantages: it reduces memory consumption compared to traditional ray tracing, and it supports micro-meshes with varying subdivision levels, further decreasing memory usage relative to micro-meshes with a uniform subdivision level.

6.1. Source code

The (source code of the) ray tracer can be found on GitHub. An application is provided where one can inspect the ray-traced micro-mesh.

6.2. Limitations and Future Work

The proposed approach is subject to several limitations that may be addressed in future work.

6.2.1. Misaligning micro-triangles

A key issue arises with micro-meshes exhibiting varying subdivision levels: the reconstructed micro-triangles do not always coincide with those of the fully tessellated mesh. As outlined in Section 3.5, when one (micro-)vertex is missing, there exist two possible subdivision patterns for constructing the micro-triangles of the next level. Since there is no canonical choice during the mesh-to-micro-mesh conversion, both variants may appear in the fully tessellated micro-mesh. However, the proposed approach consistently applies only a single subdivision pattern, which can cause discrepancies between the generated micro-mesh and the fully tessellated micro-mesh by their method [16]. While such differences are generally not visible in cases where shading relies solely on (colored) lighting or shadows, they may lead to visible artifacts when, for example, textures are applied. In particular, mismatched subdivision patterns can yield inconsistent interpolated attributes such as UV coordinates, causing incorrect texture sampling.

Future research could address this limitation by reconstructing micro-triangles in a manner that guarantees consistency with the fully tessellated micro-mesh representation by Maggiordomo et al. [16].

6.2.2. Inefficiencies in Displacement Scale Storage

A second limitation concerns memory usage in micro-meshes with varying subdivision levels. As discussed in Section 4, displacement scales are stored for all micro-vertices using a grid-based coordinate system, which enables direct indexing into the buffer. However, when certain micro-vertices are unused due to differing subdivision levels in neighboring base triangles, they do not possess displacement scales. This reduces memory consumption, but also disrupts the storage order. Since buffer indices are computed directly from grid coordinates, this misalignment can lead to invalid indexing. For example, in Figure 4.1, if the micro-vertex with coordinate $(3, 3)$ is omitted, subsequent entries shift left, reducing the buffer size to 14. This shift, however, results in out-of-bounds access when retrieving the displacement scale at $(4, 4)$ and incorrect lookups for coordinates such as $(4, 0)$.

To avoid these issues, the current approach stores a dummy displacement scale for unused micro-vertices. While this strategy preserves ordering and indexing consistency, it leads to unnecessary memory overhead. Future research could explore alternative grid-based indexing schemes that eliminate the need to store displacement scales for unused micro-vertices, thereby reducing memory consumption.

6.2.3. Animation

The ray tracer developed in this thesis is currently limited to static meshes. In practice, however, animated meshes are widely used in numerous applications. Gruen et al. [8], whose work served as a significant inspiration for this thesis, have already introduced an approach that supports animated micro-meshes.

Future research could extend the presented ray tracer to incorporate animated micro-meshes, thereby broadening its applicability.

6.2.4. Extension to RTX Mega Geometry

Following the deprecation of NVIDIA's micro-meshes [7], future research could explore adapting the proposed approach to operate with their successor, RTX Mega Geometry. At a fundamental level, both representations share conceptual similarities: micro-meshes are defined in terms of base triangles and micro-triangles, whereas RTX Mega Geometry employs clusters of triangles. Consequently, the presented method could, in principle, be extended to support RTX Mega Geometry, although additional research would be necessary to address the differences in data organization and processing.

References

[1] Tamy Boubekeur and Marc Alexa. "Phong tessellation". In: *ACM SIGGRAPH Asia 2008 papers*. 2008, pp. 1–5. DOI: https://doi.org/10.1145/1409060.1409094.

[2] Jiawen Chen et al. "Real-time volumetric shadows using 1d min-max mipmaps". In: *Symposium on Interactive 3D Graphics and Games*. 2011, pp. 39–46. DOI: https://doi-org.tudelft.idm.oclc.org/10.1145/1944745.1944752.

[3] Paolo Cignoni et al. "MeshLab: an Open-Source Mesh Processing Tool". In: *Eurographics Italian Chapter Conference*. Ed. by Vittorio Scarano, Rosario De Chiara, and Ugo Erra. The Eurographics Association, 2008. ISBN: 978-3-905673-68-5. DOI: 10.2312/LocalChapterEvents/ItalChap/ItalianChapConf2008/129-136.

[4] Robert L Cook. "Shade trees". In: *Proceedings of the 11th annual conference on Computer graphics and interactive techniques*. 1984, pp. 223–231. DOI: https://doi.org/10.1145/964965.808602.

[5] Kirill Dmitriev and Yury Uralsky. "Soft shadows using hierarchical min-max shadow maps". In: *Game Development Conference*. 2007.

[6] Yishun Dou et al. "Differentiable Micro-Mesh Construction". In: *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 2024, pp. 4294–4303. DOI: https://doi.org/10.1109/CVPR52733.2024.00411.

[7] NVIDIA GameWorks. *Displacement Micro-Map Toolkit*. Accessed: 2025-10-04. 2023. URL: https://github.com/NVIDIAGameWorks/Displacement-MicroMap-Toolkit.

[8] Holger Gruen et al. "Ray Tracing Animated Displaced Micro-Meshes". In: *Computer Graphics Forum*. Vol. 43. 7. Wiley Online Library. 2024, e15225. DOI: https://doi.org/10.1111/cgf.15225.

[9] Igor Guskov et al. "Normal meshes". In: *Proceedings of the 27th annual conference on Computer graphics and interactive techniques*. 2000, pp. 95–102. DOI: https://doi.org/10.1145/344779.344831.

[10] Victor Klee and Michael C Laskowski. "Finding the smallest triangles containing a given convex polygon". In: *Journal of Algorithms* 6.3 (1985), pp. 359–375.

[11] Manuel Kraemer, Dylan Lacewell, and Ardavan Kanani. "Real Time Path-Tracing with NVIDIA RTX MegaGeometry". In: *Proceedings of the Special Interest Group on Computer Graphics and Interactive Techniques Conference Real-Time Live!* 2025, pp. 1–2. DOI: https://doi-org.tudelft.idm.oclc.org/10.1145/3721243.3735983.

[12] Christoph Kubisch et al. *NVIDIA RTX Mega Geometry Now Available with New Vulkan Samples*. Accessed: 2025-10-04. 2025. URL: https://developer.nvidia.com/blog/nvidia-rtx-mega-geometry-now-available-with-new-vulkan-samples/.

[13] Oliver Laric. *Three D Scans - Free 3D scan archive*. https://threedscans.com/. 2012.

[14] Aaron Lee, Henry Moreton, and Hugues Hoppe. "Displaced subdivision surfaces". In: *Proceedings of the 27th annual conference on Computer graphics and interactive techniques*. 2000, pp. 85–94. DOI: https://doi.org/10.1145/344779.344829.

[15] Alexander Lier et al. “A high-resolution compression scheme for ray tracing subdivision surfaces with displacement”. In: *Proceedings of the ACM on Computer Graphics and Interactive Techniques* 1.2 (2018), pp. 1–17. DOI: https://doi.org/10.1145/3233308.

[16] Andrea Maggiordomo, Henry Moreton, and Marco Tarini. “Micro-mesh construction”. In: *ACM Transactions on Graphics (TOG)* 42.4 (2023), pp. 1–18. DOI: https://doi.org/10.1145/3592440.

[17] Nadia Magnenat-Thalmann, Richard Laperrière, and Daniel Thalmann. “Joint-dependent local deformations for hand animation and object grasping”. In: *Proceedings on Graphics interface'88*. 1989, pp. 26–33.

[18] Microsoft. *DirectX Raytracing (DXR)*. Accessed: 2025-10-02. URL: https://microsoft.github.io/DirectX-Specs/d3d/Raytracing.html.

[19] NVIDIA. *Micro-Mesh – Basics*. NVIDIA developer presentation. Accessed: 2025-09-27. URL: https://developer.download.nvidia.com/ProGraphics/nvpro-samples/slides/Micro-Mesh_Basics.pdf.

[20] NVIDIA. *Micro-Mesh Graphics Primitive for Micro Triangles*. 2022. URL: https://developer.nvidia.com/rtx/ray-tracing/micro-mesh (visited on 02/02/2025).

[21] NVIDIA Corporation. *NVIDIA Ada GPU Architecture Whitepaper*. 2023. URL: https://images.nvidia.com/aem-dam/Solutions/geforce/ada/nvidia-ada-gpu-architecture.pdf.

[22] NVIDIA Corporation. *NVIDIA Turing Architecture Whitepaper*. Tech. rep. NVIDIA, 2018. URL: https://images.nvidia.com/aem-dam/en-zz/Solutions/design-visualization/technologies/turing-architecture/NVIDIA-Turing-Architecture-Whitepaper.pdf.

[23] Joseph O'Rourke et al. “An optimal algorithm for finding minimal enclosing triangles”. In: *Journal of Algorithms* 7.2 (1986), pp. 258–269.

[24] VV Sanzharov et al. “Examination of the Nvidia RTX”. In: *CEUR Workshop Proceedings*. Vol. 2485. 2019, pp. 7–12.

[25] Simon Scheckel and Andreas Kolb. “Min-max mipmaps for efficient 2d occlusion culling”. In: *Conference on Computer Graphics, Visualization and Computer Vision*. 2016, pp. 13–16.

[26] Simplygon. *Micro-Meshes*. Accessed: 2025-10-02. URL: https://www.simplygon.com/features/micromeshes.

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

[28] Martin Stich. *Introduction to NVIDIA RTX and DirectX Ray Tracing*. NVIDIA Technical Blog. Mar. 2018. URL: https://developer.nvidia.com/blog/introduction-nvidia-rtx-directx-ray-tracing/.

[29] Theo Thonat et al. “Tessellation-Free Displacement Mapping for Ray Tracing”. In: 40.6 (Dec. 2021). ISSN: 0730-0301. DOI: 10.1145/3478513.3480535. URL: https://doi.org/10.1145/3478513.3480535.

[30] Alex Vlachos et al. “Curved PN triangles”. In: *Proceedings of the 2001 symposium on Interactive 3D graphics*. 2001, pp. 159–166. DOI: https://doi.org/10.1145/364338.364387.