Micro-Mesh - Basics
Opacity & Displacement Micro-Meshes
Micromeshes and Micromaps
Micromeshes and Micromaps

Micromeshes
Micromeshes are created through fixed subdivision of input triangles. Each level evenly splits a triangle into 4 triangles.

Micromaps
Micromaps store per-microvertex or per-microtriangle values for these subdivided triangles. The values can be block-compressed. Each subdivided input triangle has its own set of values, there is no values sharing between the coarse input triangles (aka base triangles). There is no need for UV coordinates as the mapping from the triangle UV to the storage index is handled through a spatial curve. But it is sensitive to triangle vertex ordering.
Micromeshes and Micromaps

“Bird curve” a new spatial indexing curve in barycentric space

Here we focus on the ordering of microtriangles, later in the displacement compression chapter the microvertex ordering is illustrated. The center image shows the recursive nature of the winding and orientation changes as well.
Micromeshes and Micromaps

“Bird curve” Recursive Splitting Rule

An input triangle is split into four sub-triangles with the following logic (this can be recursive / hierarchical):

0: W, WU, VW

1: VW, UV, WU (flips source winding)

2: WU, U, UV

3: UV, VW, V (flips source winding)

Triangle (W,U,V) is split into four children
Micromaps

Micromaps represent scalar values stored on a barycentric grid for each triangle. It is a per-triangle map that isn’t sampled (i.e. not like a texture) but uses unfiltered fetches. The triangle’s vertex ordering is critical to this addressing, given it is based on barycentric coordinates.

Mesh Triangle Mapping

Each mesh-triangle is mapped to a micromap-triangle

This is typically 1:1 mapping, although a dedicated mapping buffer can exist to allow re-use
Micromaps

Micromap Storage
Micromap storage contains information for many triangles and their values.

```cpp
Micromap {  
    MicromapTriangle triangles[];  
    ValueType values[];  
    ValueFormat valueFormat;  
    // per-microvertex / microtriangle  
    ValueFrequency valueFrequency;  
    // value layout within triangle  
    ValueLayout valueLayout;  
    ...  
};
```

Micromap Triangle
- `valuesOffset` (starting location)
- `subdivisionLevel` (resolution of this triangle)
- `blockFormat` (only for compression)

Value layout (spatial curve) determines index within triangle (layout is implicit for compression).

Value layout (spatial curve) determines index within triangle (layout is implicit for compression).
Micromaps

Pseudo code for illustrative purposes only

Micromaps

Micromap {
    MicromapTriangle triangles[];
    ValueType values[];
    ... // some more meta info
};

MicromapTriangle {
    U32 valuesOffset;
    U16 subdivisionLevel;
    U16 blockFormat;
};

Mesh Triangle Mapping

// optional mapping buffer allows re-use of map data for different mesh triangles


Value Fetching for a provided microvertex (uncompressed)

ValueType* triangleValues = micromap.values[ micromapTriangle.valuesOffset ];
// index into triangleValues using the spatial storage layout based on microvertex coordinates

ValueType microVertexValue = triangleValues[ getLayoutIndex(micromap.valueLayout, microvertex.barycentricIntegerUV ) ];
Micromaps

Micromap Block Compression

Values can be compressed. In this case the triangle is split into block triangles, each using one compressed block. The ordering of blocks and the splitting is part of the compression scheme (see later)

```cpp
MicromapTriangle {
    // starting location
    U32 valuesOffset;
    // resolution of this triangle
    U16 subdivisionLevel;
    // only for compression
    U16 blockFormat;
};

// implicit information based on
// compression scheme, not stored
BlockTriangle {
    // position within parent triangle
    BaryUV vertexUVs[3];
    // block start within parent values
    U32 byteOffset;
    ... // some more meta info
};
```
Micromaps

Pseudo code for illustrative purposes only

Micromaps

```cpp
Micromap {
    MicromapTriangle triangles[];
    Byte values[];
    ...
}; // some more meta info

MicromapTriangle {
    U32 valuesOffset;
    U16 subdivisionLevel;
    U16 blockFormat;
};

BlockTriangle {
    BaryUV vertexUVs[3]; // position within parent map triangle
    U32 byteOffset; // block start within parent values
    ...
}; // some more meta info

Value Fetching for a provided microvertex (compressed)

// compressed or specially packed values operate with byte offsets
Bytes* triangleValues = &micromap.values[ micromapTriangle.valuesOffset ];

// find which block based on microvertex uv
BlockTriangle blockTriangle = getBlockTriangle( micromapTriangle.subdivisionLevel, micromapTriangle.blockFormat,
    microvertex.barycentricIntegerUV);

ValueType uncompressedBlock[];

decompress(uncompressedBlock, micromapTriangle.subdivisionLevel, micromapTriangle.blockFormat, triangleValues + blockTriangle.byteOffset);

// use blockTriangle.vertexUVs to convert between map and block triangle coordinate space
ValueType microVertexValue = uncompressedBlock[ getLocalBlockIndex(blockTriangle, microvertex.barycentricIntegerUV ) ];
```
Micromaps Conclusion

Micromaps

The values stored in micromaps can be per-microvertex or per-microtriangle (result of subdivision). The ordering of values within a triangle is defined through a layout (we standardized two for now)

The raytracing APIs will support:

- **Displacement Micromap (DMM)** per-microvertex scalar displacement
- **Opacity Micromap (OMM)** per-microtriangle opacity

The containers and operations of the SDK are versatile enough to store any other attributes. This is useful to experiment with storing other shading attributes, such as normals in micromaps, and fetching them in shaders manually.

**However**, compression/special packing only exists for scalar displacement (unorm11) and opacity (mix of uint1/uint2) and raytracing APIs only accept those special formats.
Opacity Micromaps
Opacity

Opacity Micromap

Accelerate ray-tracing of transparent surfaces by reducing any-hit shader invocations and use micromeshes as visibility mask (independent of displacement, can use much higher subdivisions)

Encode visibility per Microtriangle (1 or 2 bits)

- values stored along “bird curve”
- 1 bit: 0 miss, 1 hit
- 2 bit: 0 miss, 1 and 2 “unknown states”, 3 hit

Reduced 2 bit visibility (effective 2x2 bits per low-res texel)
Opacity

Opacity Micromap

Dynamically map the “unknowns” to either any-hit shader, miss or hit at trace time.

- Reduced 2 bit visibility can be dynamically mapped.
- Soft shadow trace uses binary mapping.
- Reflection trace maps unknowns to any-hit shader.
Displacement Micromaps
Displacement

New representation of high geometric detail

Uses fixed power of two subdivision pattern on base-triangles and scalar displacements in barycentric space

Encodes the scalar values hierarchically along a spatial curve to get good compression and locality

Reduces BVH build time (lo-res base mesh, mostly static pre-computed displacement data) and memory consumption significantly
Displacement

Geometry representation

Each base triangle has a user-defined power of two fixed subdivision. An adjacent triangle may differ in subdivision by a factor of two along the edge and watertightness can still be preserved.

Corner displacement directions are provided as fp16 vectors per base vertex and are linearly interpolated across.

Scalar displacements are provided in barycentric space per micro vertex
Displacement

Base Mesh

Micromeshes

Displaced Micromeshes

Model courtesy of Autodesk, armor part from “Turtle Barbarian” by Jesse Sandifer
Displacement

Geometry representation

The API will support base triangle subdivision of up to level 5 == 1024 Micro Triangles for displacement

<table>
<thead>
<tr>
<th>Subdivision Level</th>
<th>Micro Triangles</th>
<th>Micro Vertices</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>1</td>
<td>4</td>
<td>6</td>
</tr>
<tr>
<td>2</td>
<td>16</td>
<td>15</td>
</tr>
<tr>
<td>3</td>
<td>64</td>
<td>45</td>
</tr>
<tr>
<td>4</td>
<td>256</td>
<td>153</td>
</tr>
<tr>
<td>5</td>
<td>1024</td>
<td>561</td>
</tr>
</tbody>
</table>
Displacement

Watertight representation

For each base triangle the edges that require decimation can be marked using one uint8. The first three bits represent the edges \((v_0, v_1)\) \((v_1, v_2)\) \((v_2, v_0)\) for a triangle defined as \((v_0, v_1, v_2)\).

The intersection will perform as if triangles along the edge are adjusted accordingly, or by other means in the hw that ensure watertight results.

The baker and the encoder ensure that appropriate scalar values match
Displacement

Watertight representation

For each base triangle the edges that require decimation can be marked (one uint8 per triangle, first 3 bits used).

The intersection will perform as if triangles along the edge are adjusted accordingly.

The baker and the encoder ensure that appropriate scalar values match.
Linear Displacement

```
microvertex.position =
interpolate(base.positions[], microvertex.barycentric) +
// note, no normalization of directions!
interpolate(base.directions[], microvertex.barycentric)
// unorm11 displacement value in [0,1]
* (microvertex.displacement);
```

As result all displacements are within a prismoid shape (shell volume) that is created by the minimum (position) and maximum (position + direction) triangle.
Displacement - Fitting

**Global** bias/scale can be suboptimal

Spikes in the displacement value range mean less precision of unorm11 overall and larger shell volume. Larger shell volume affects tracing performance negatively.

**Local** per-vertex bias/scale preferred

Compute per-vertex direction bounds (bias, scale) along the original vectors that fit the data. They allow adjusting position & direction vectors.

These bounds can be provided at BLAS buildtime:

```
blas_position = position + direction * bounds.bias
blas_direction = direction * bounds.scale
```

**Caveat:** the BLAS returns the adjusted values in intrinsics

Simplified 2D section view of connected triangles, illustrates reduced shell volume by fitted bounds
Displacement - Fitting

Shell volume using global bias & scale

Yellow: max triangle
Cyan: min triangle

Per-base-vertex bias & scale yields a $\frac{1}{4}$ of the original total shell volume

Approximately 5 x speed-up for primary rays (different view, many instances)
Displacement - Fitting Side Effect

The fitting process requires resampling of displacements as positions and directions are affected by the fitted bounds.

The projected point onto base triangle might drift, changing UV/texcoord interpolation a bit. The result benefits from rebaking of textures or use of indirection textures... Content dependent issue. If done early in asset pipeline, no need to rebake etc.
Displacement Compression
Displacement Compression

Hierarchical Delta Compression

Reconstruction of displacement:

- Predict via averaging of split edge vertices
- Apply signed delta correction value.

![Diagram showing vertex subdivision levels and signed correction](attachment:image.png)
Microvertex Order

Vertex order within subdivided triangle

The “bird curve” order for microvertices is also based on hierarchical splitting, and stores vertices in groups of triplets.

Microvertex values are stored in triplets.

Example for subdivision level 2:
16 microtriangles, 15 microvertices

Triplets are ordered breadth first by subdivision level. Within a level they follow the bird curve order.

Resulting microvertex ordering 0…14. Ordered by subdivision level in which they were added.
Microvertex Order

Vertex order within subdivided triangle

The “bird curve” order for microvertices is also based on hierarchical splitting, and stores vertices in groups of triplets.
Compression

Hierarchical Delta Compression
Correction values are stored in “bird curve” order within compressed block.

Reconstruction of displacement:
- Predict via averaging of split edge vertices
- Apply signed delta correction value.

\[
d(2) = \text{correction}(2)
\]

Initial anchors are lossless

\[
d(4) = \frac{(d(2) + d(1) + 1)}{2} + \text{correction}(4)
\]

\[
d(7) = \frac{(d(5) + d(3) + 1)}{2} + \text{correction}(7)
\]
Displacement Compression

Block Layout

Correction values are stored in 512 or 1024 bit blocks.
Blockformat-dependent width of bits per subdivision
“shift bits” allow to adjust the amplitude of corrections (similar to shared exponent, see next slide)
Displacement Compression

Block Layout

Vertices are classified as interior or on micromesh edges and shift bits are used to adjust their correction value:

\[
\text{correction}(vtx) = \text{correction}(vtx) \ll \text{shiftBits}[vtx\text{SubdivLevel}][vtx\text{Type}];
\]

<table>
<thead>
<tr>
<th>Shift Bits</th>
<th>.... 1023 / 511 bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
</tr>
</tbody>
</table>

- Stored reversed, highest level in lower bits
- Different shift values for each vertex type

Example for highlighted vertices with vtxSubdivLevel 2

- vtxType 0: Interior
- vtxType 1: Edge 0
- vtxType 2: Edge 1
- vtxType 3: Edge 2
Displacement Compression

Illustration Guide

The next slides will use illustrations around base-triangles and sub-triangles.

We previously used the data structure called BlockTriangle (slide 11).

Sub-Triangles == Block-Triangles, terms can be used interchangeably. One sub-triangle is always represented by one compressed block.
Displacement Compression

Compression & Sub-Triangles

Similar to BCx compression we establish block-compressed formats that can represent displaced micro triangles at different fidelity in **512 or 1024 bits**. Decoded to **unorm11**.

```cpp
enum BlockFormatDispC1 {
    // displacement for
eR11_unorm_lvl3_pack512 = 1, // 64 microtriangles (45 x uncompressed u11)
eR11_unorm_lvl4_pack1024 = 2, // 256 microtriangles
eR11_unorm_lvl5_pack1024 = 3, // 1024 microtriangles
};
```

**Base-triangles** can be split **uniformly** into many **sub-triangles** to represent larger resolutions or higher quality formats. One sub-triangle is represented by one compressed block.
Displacement Compression

Compression Efficiency

Each block stores displacement values for its microtriangles independent of other blocks.

This can reduce overall compression a bit.

Favorable to attempt to use a single block per base-triangle.

The smallest block format is 512 bit and allows storing the 45 displacement values for 64 microtriangles in uncompressed unorm11.

That means using less subdivision than level 3 (equivalent to 64 microtriangles) will waste memory.
Displacement Compression

Compression & Sub-Triangles

Triangles are split using the “bird curve” rule seen before. They are stored hierarchically in depth first order:

0: W, WU, VW
1: VW, UV, WU (flips source winding)
2: WU, U, UV
3: UV, VW, V (flips source winding)

These winding and orientation changes add a bit of complexity in the encoding/decoding process
Example
Example

Facet-shaded and colored by sub-triangle

24% compared to 16-bit uncompressed input

Different compression formats are used

Wireframe:
Lo-res base mesh
1024 x expansion:
7.5 K triangles to 7.7 M

Content-dependent compression:
White: 512bit per 64 microtriangles
Green: 1024bit per 256 microtriangles
Orange: 1024 bit per 1024 microtriangles
Additional Links & Information

These slides are part of the NVIDIA Micro-Mesh SDK

Relevant Repositories
https://github.com/NVIDIAGameWorks/Opacity-MicroMap-SDK
https://github.com/NVIDIAGameWorks/Displacement-MicroMap-SDK
https://github.com/NVIDIAGameWorks/Displacement-MicroMap-Toolkit
https://github.com/NVIDIAGameWorks/Displacement-MicroMap-BaryFile

Support Contacts for SDK
opacitymicromap-sdk-support@nvidia.com
DisplacedMicroMesh-SDK-support@nvidia.com