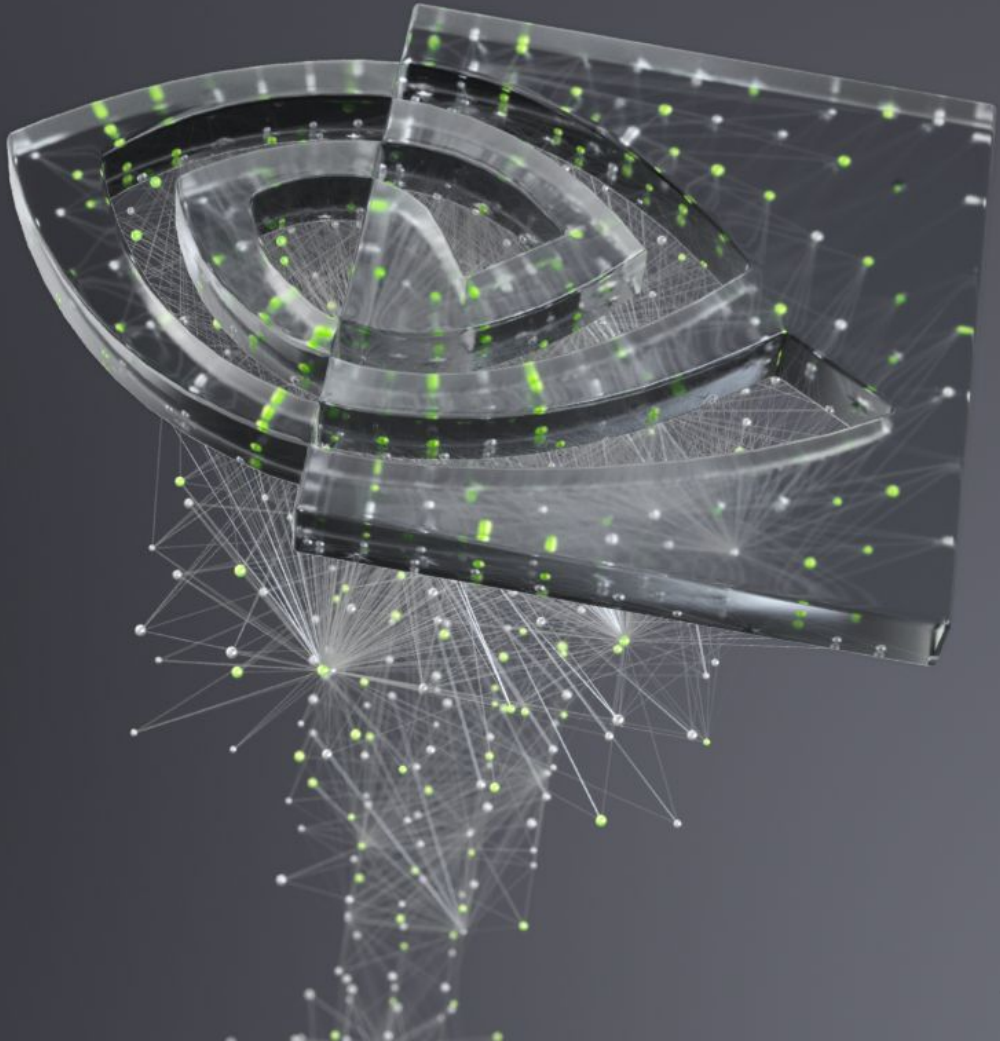
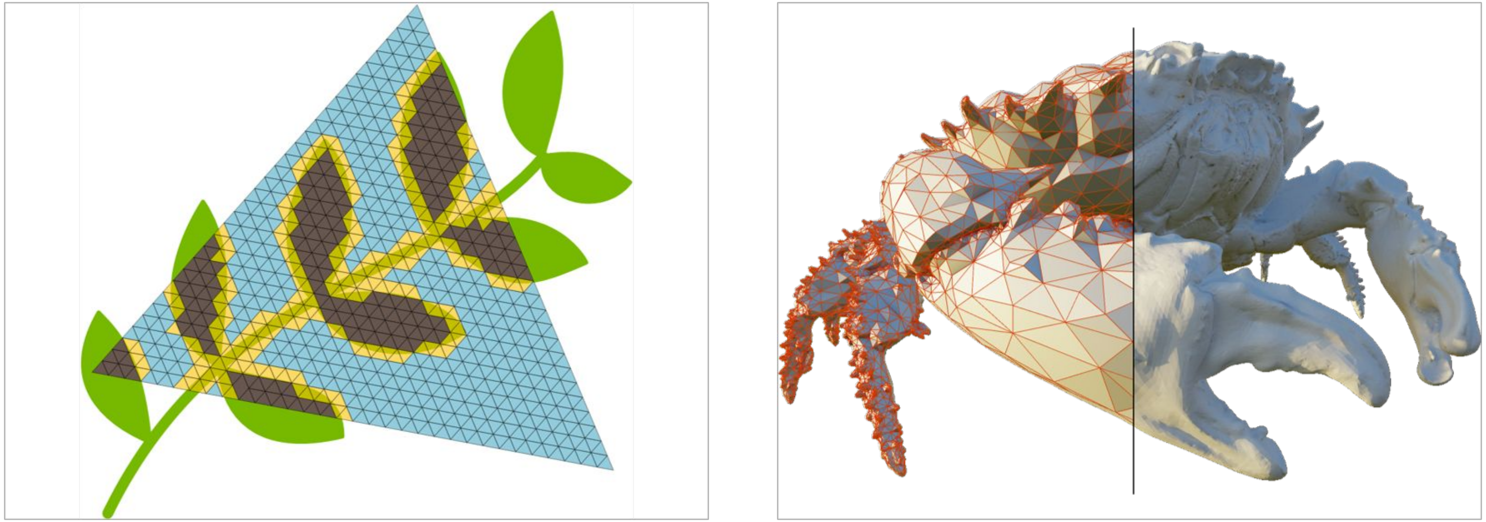
Micro-Mesh - Basics





Opacity & Displacement Micro-Meshes

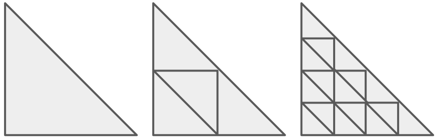
Micromeshes and Micromaps

Micromeshes and Micromaps

Micromeshes

Micromeshes are created through fixed subdivision of input triangles.

Each level evenly splits a triangle into 4 triangles.



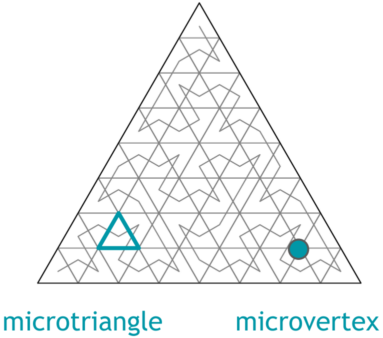
Micromesh result for
subdivision level 0, 1 and 2

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).

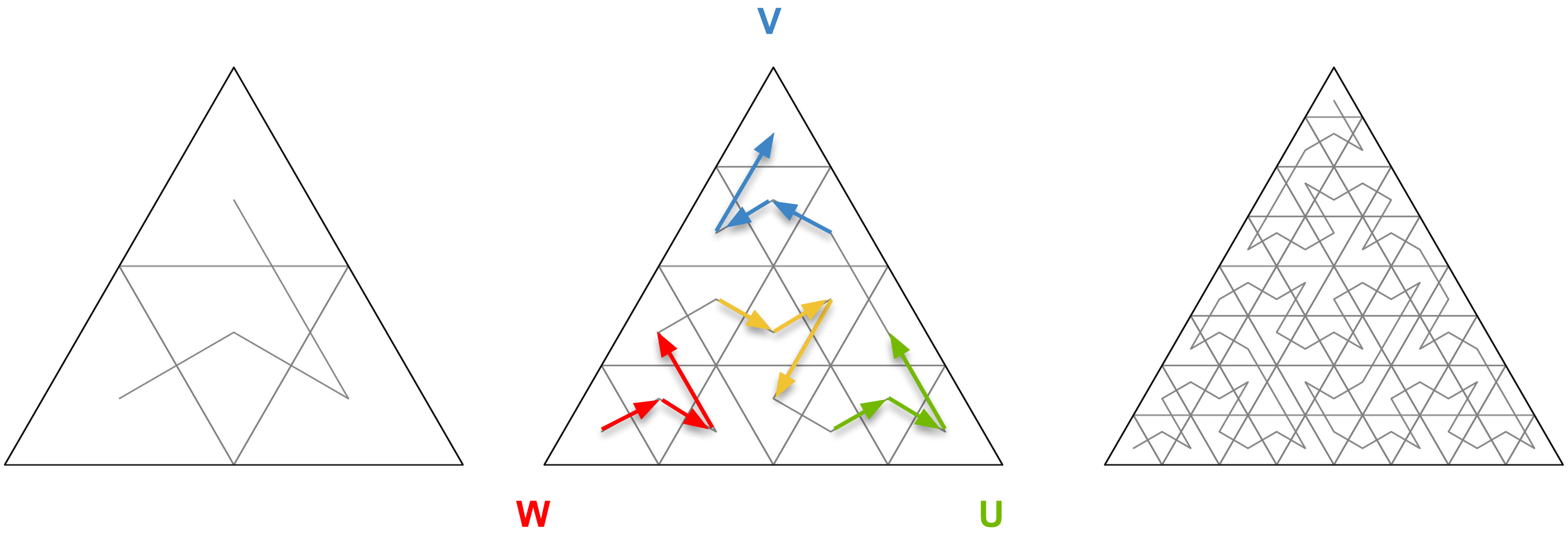
There is no need for UV coordinates as the mapping from the triangle UV to the storage index is handled through a spatial curve. But it is sensitive to triangle vertex ordering.



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.

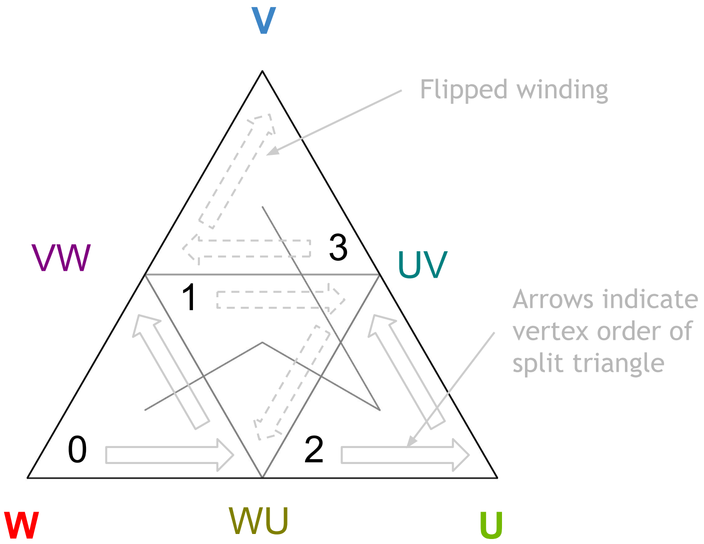


Micromeshes and Micromaps

“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

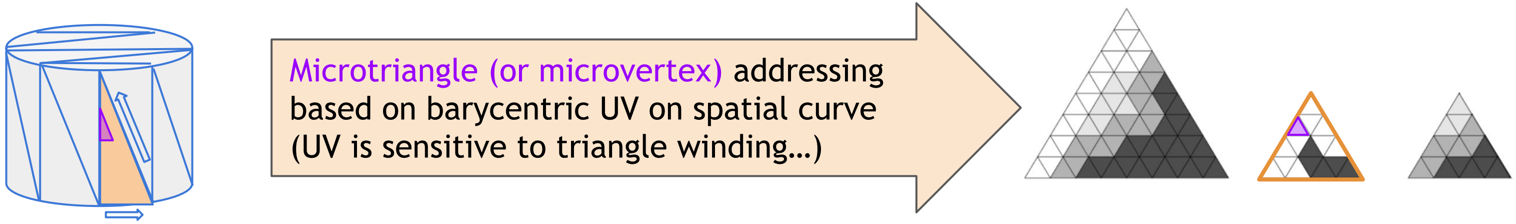
Micromaps

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

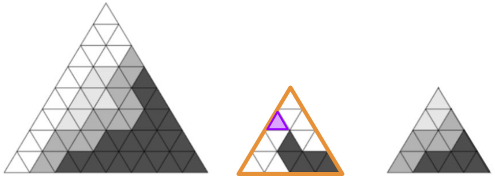


Microtriangle (or microvertex) addressing based on barycentric UV on spatial curve (UV is sensitive to triangle winding…)

3D Mesh contains mesh triangles, mapped to micromap triangle

Micromap contains subdivision info and values for the subdivided triangles

Micromaps


Micromap Value Illustration

Micromap Storage

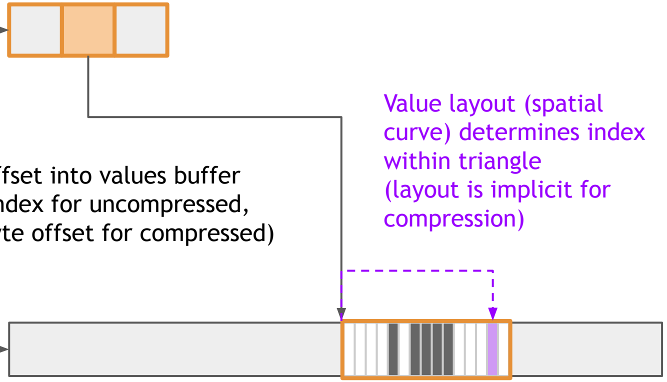
Micromap storage contains information for many triangles and their values

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

```
Micromap {
  MicromapTriangle triangles[];
  ValueType        values[];

  ValueFormat      valueFormat;
  // per-microvertex / microtriangle
  ValueFrequency   valueFrequency;
  // value layout within triangle
  ValueLayout      valueLayout;

  ...
};
```


Micromap Triangles Buffer

Offset into values buffer
(index for uncompressed,
byte offset for compressed)

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

Micromap Values Buffer

Micromaps

Pseudo code for illustrative purposes only

Micromaps

```
Micromap {
  MicromapTriangle triangles[];
  ValueType        values[];

  ... // some more meta info
};
```

```
MicromapTriangle {
  U32 valuesOffset;
  U16 subdivisonLevel;
  U16 blockFormat;
};
```

Mesh Triangle Mapping

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

MicromapTriangle micromapTriangle = mappings ? micromap.triangles[ mappings [ meshTriangleIndex ] ]
                                             : micromap.triangles[ meshTriangleIndex ];
```

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

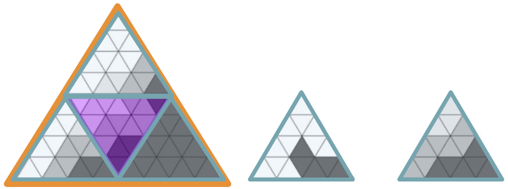
Micromap Block Compression

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)

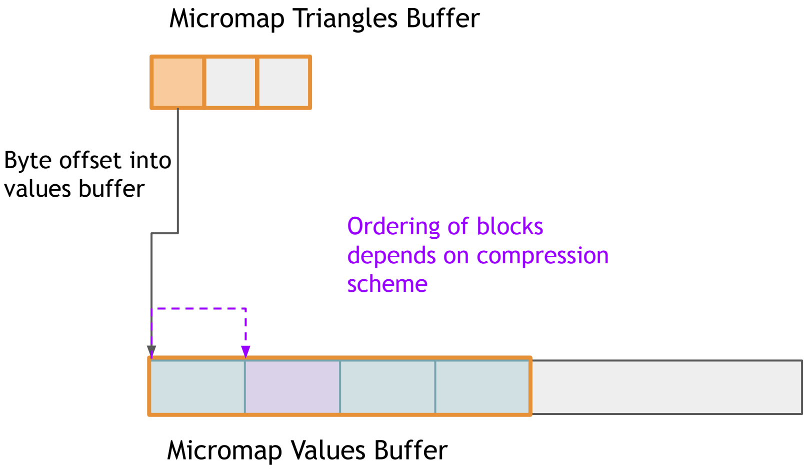
```
MicromapTriangle {
  // starting location
  U32 valuesOffset;
  // resolution of this triangle
  U16 subdivisonLevel;
  // only for compression
  U16 blockFormat;
};
```

```
// 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
};
```



Micromap Triangle split into one or more compressed blocks / block triangles



Micromap Triangles Buffer

Byte offset into
values buffer

Ordering of blocks
depends on compression
scheme

Micromap Values Buffer

Micromaps

Pseudo code for illustrative purposes only

Micromaps

```
Micromap {
  MicromapTriangle triangles[];
  Byte             values[];

  ... // some more meta info
};
```

```
MicromapTriangle {
  U32 valuesOffset;
  U16 subdivisonLevel;
  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 gives us information about the local position of the block within the base triangle, as well as byte offsets
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

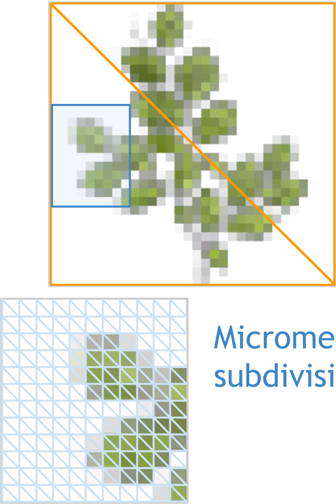
Opacity Micromap

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)

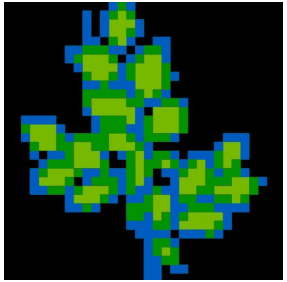
Encode visibility per Microtriangle (1 or 2 bits)

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

Original Texture



Micromesh subdivision



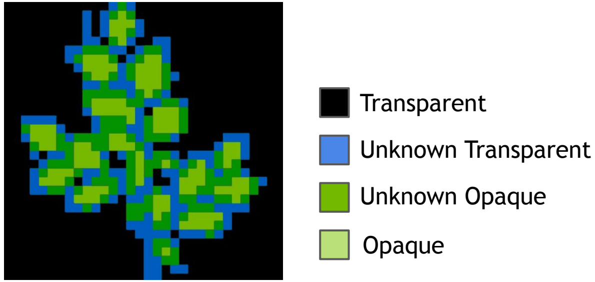
⬛ Transparent
🟦 Unknown Transparent
🟩 Unknown Opaque
▩ Opaque

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

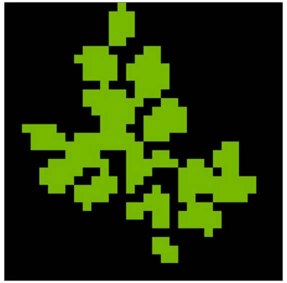
Opacity

Opacity Micromap

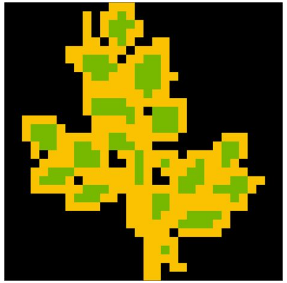
Dynamically map the “unknowns” to either any-hit shader, miss or hit at trace time.



Reduced 2 bit visibility can be dynamically mapped



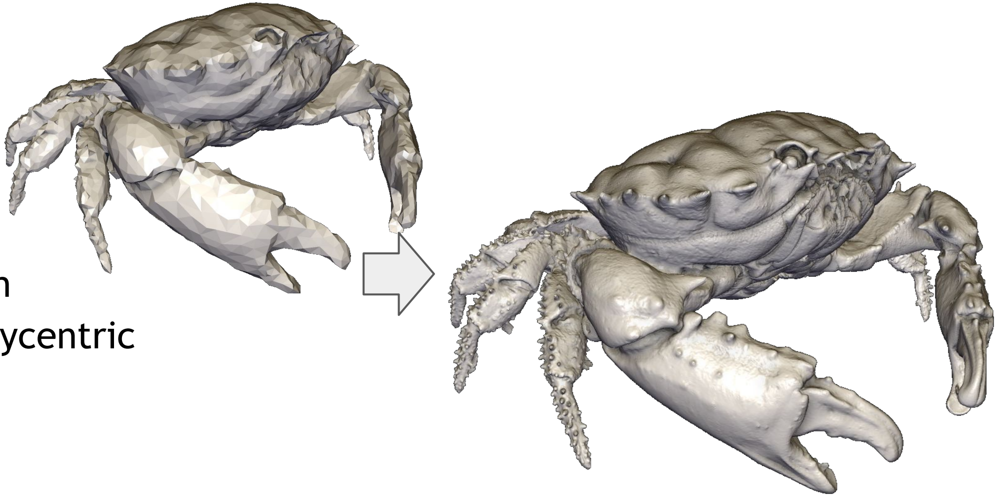
Soft shadow trace uses binary mapping



Reflection trace maps unknowns to any-hit shader

Displacement Micromaps

Displacement



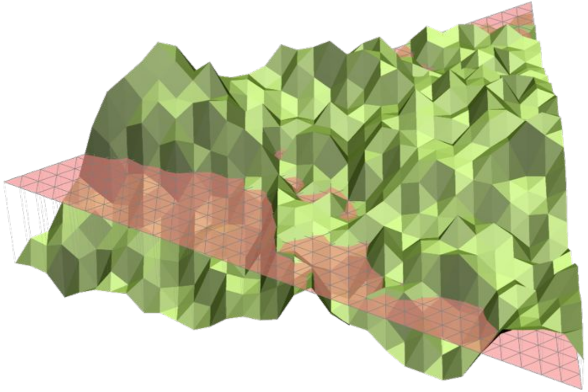
Model courtesy of threedscans.com

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



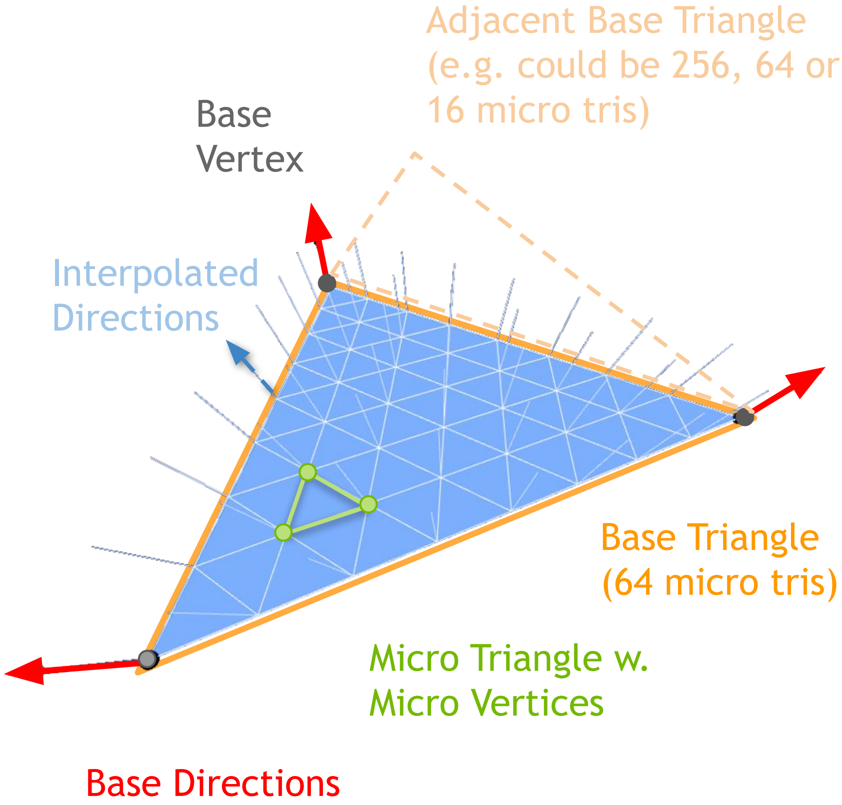
Displacement

Geometry representation

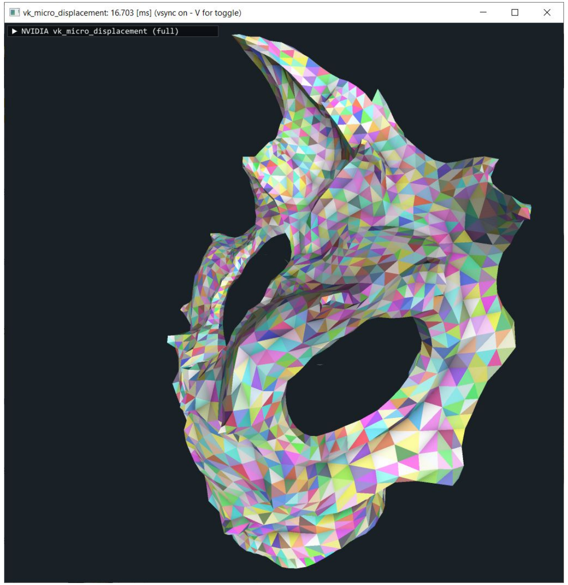
Each base triangle has a user-defined **power of two fixed subdivision**. An adjacent triangle may differ in subdivision by a factor of two along the edge and watertightness can still be preserved.

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

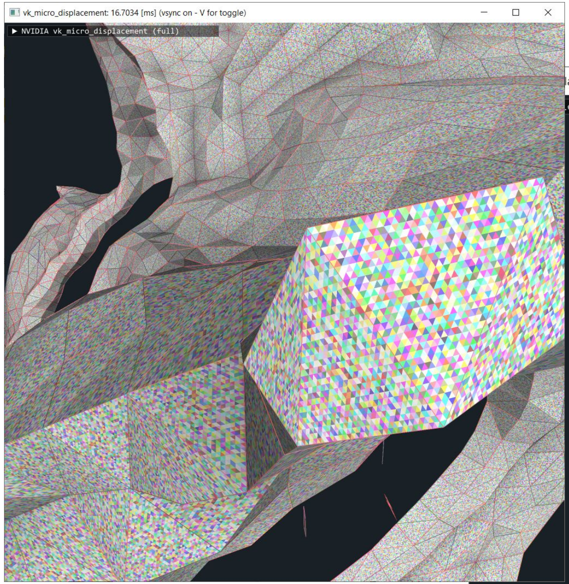
Scalar displacements are provided in barycentric space per micro vertex



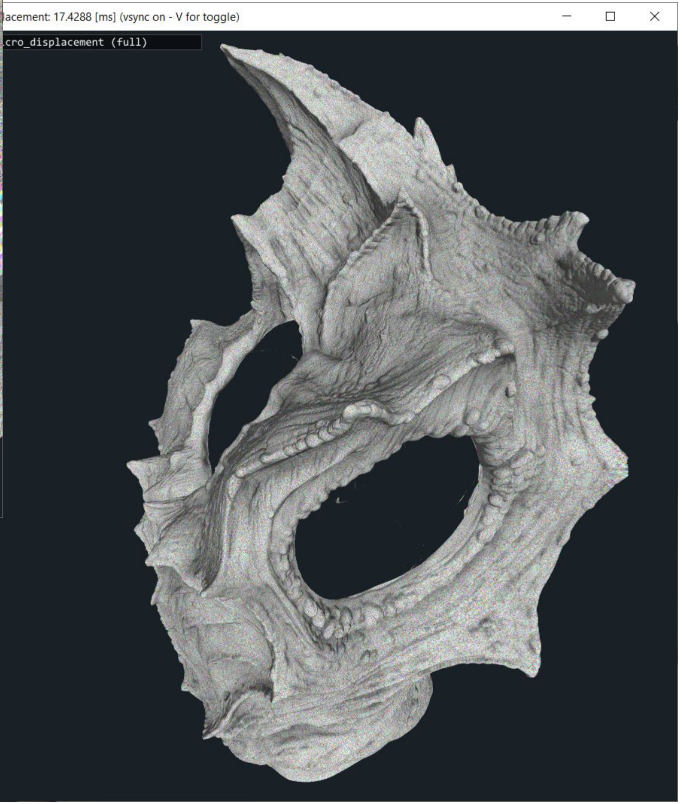
Displacement



Base Mesh



Micromeshes



Displaced Micromeshes

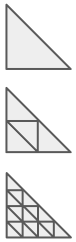
Model courtesy of Autodesk, armor part from
“Turtle Barbarian” by Jesse Sandifer

Displacement

Geometry representation

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

Subdivision Level	Micro Triangles	Micro Vertices
0	1	3
1	4	6
2	16	15
3	64	45
4	256	153
5	1024	561



Displacement

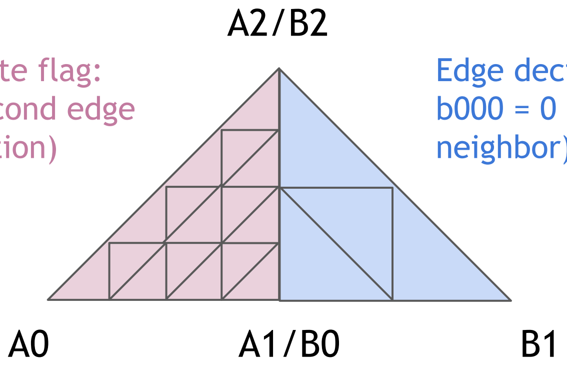
Watertight representation

For each base triangle the edges that require decimation can be marked using one uint8.

The first three bits represent
the edges (v0, v1) (v1,v2) (v2,v0)
for a triangle defined as (v0,v1,v2).

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

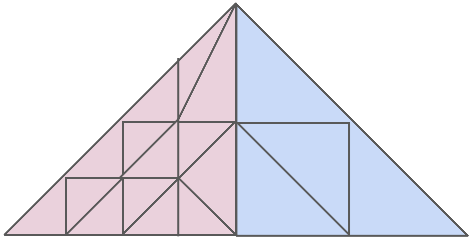


Basetriangle A
subdiv level 2

Basetriangle B
subdiv level 1

Edge decimate flag:
b010 = 2 (second edge needs reduction)

Edge decimate flag:
b000 = 0 (no reduced neighbor)



Intersection behavior
(or equivalent means)

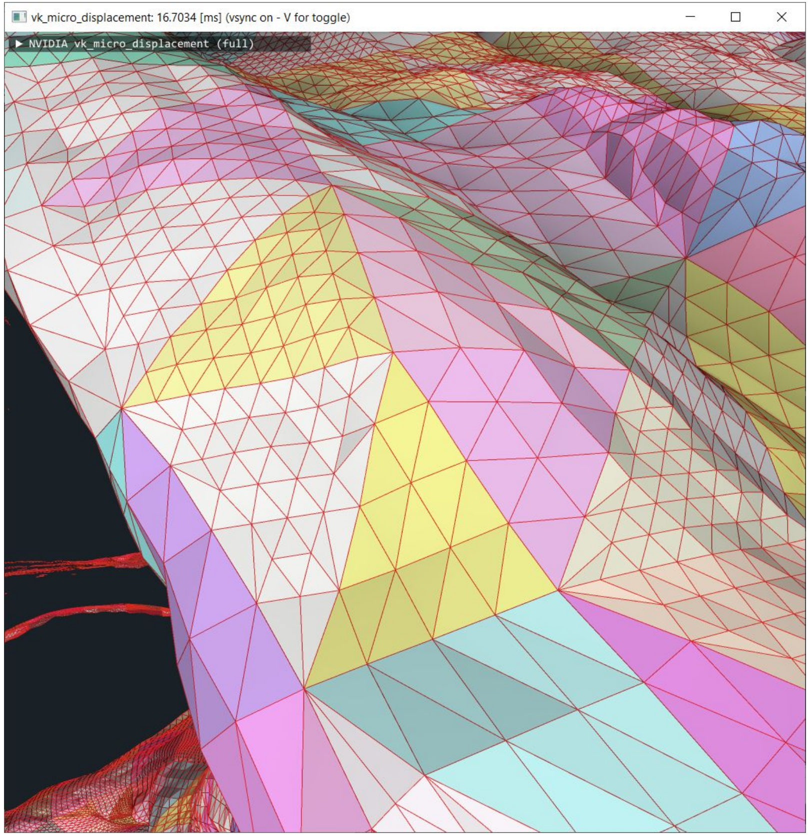
Displacement

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

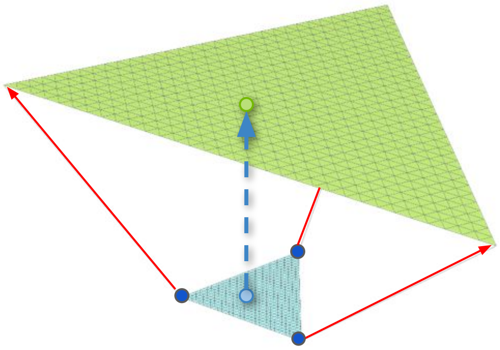
The baker and the encoder ensure that appropriate scalar values match

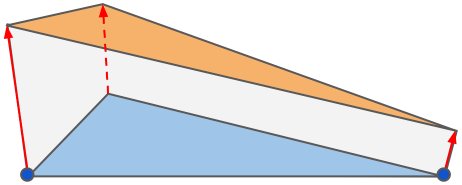


Linear Displacement

```
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.





vertex base.directions yield local max triangle

Displaced surface within min and max triangles' shell volume

vertex base.positions define the local min 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

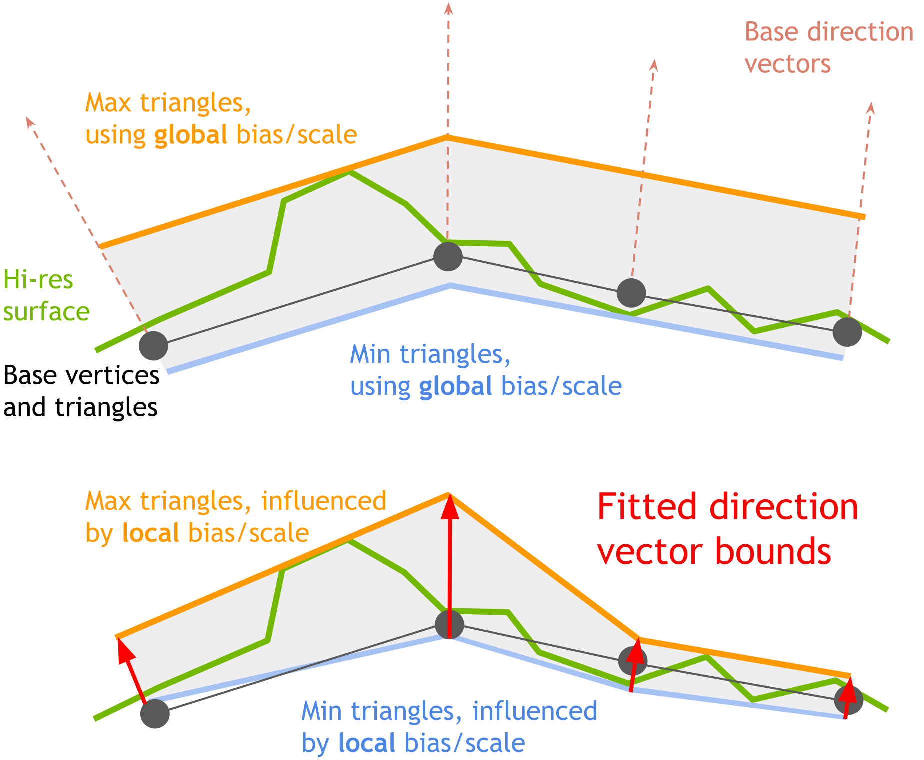
Compute per-vertex direction bounds (bias, scale) along the original vectors that fit the data. They allow adjusting position & direction vectors.

These bounds can be provided at BLAS buildtime:

blas_position = position + direction * bounds.bias

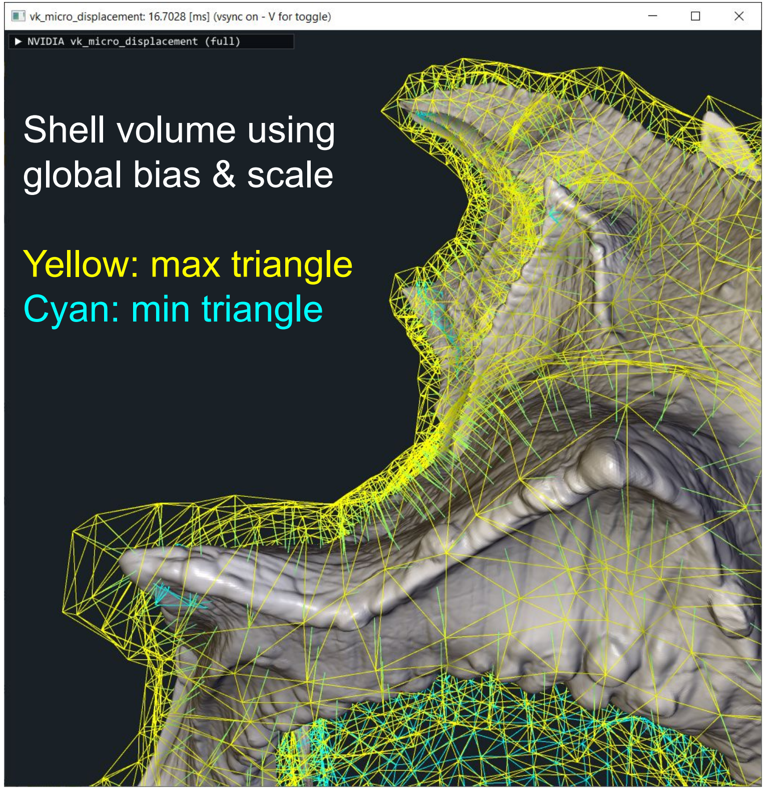
blas_direction = direction * bounds.scale

Caveat: the BLAS returns the adjusted values in intrinsics



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

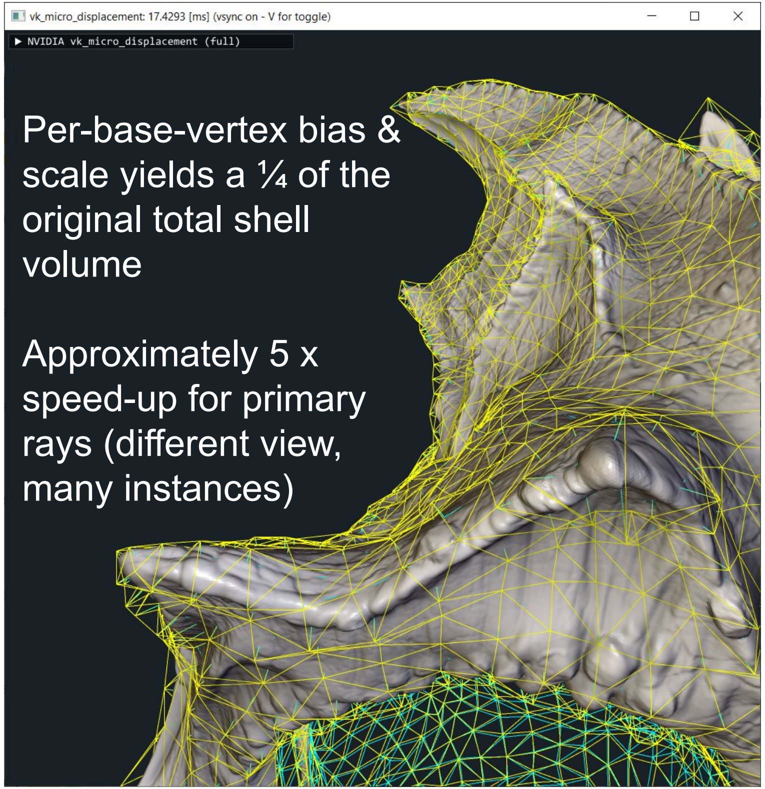
Displacement - Fitting


vk_micro_displacement: 16.7028 [ms] (vsync on - V for toggle)

▶ NVIDIA vk_micro_displacement (full)

Shell volume using global bias & scale

Yellow: max triangle
Cyan: min triangle

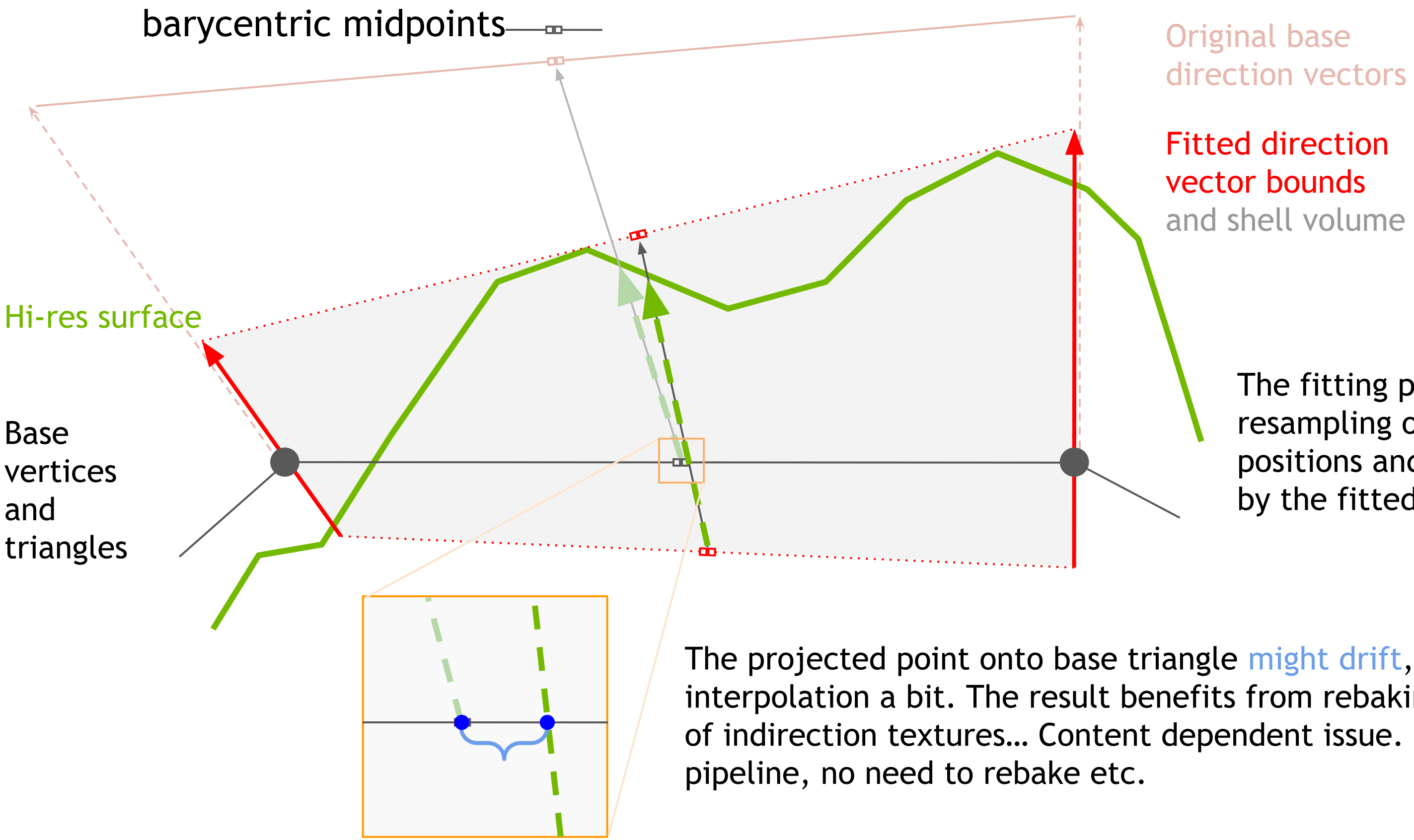

vk_micro_displacement: 17.4293 [ms] (vsync on - V for toggle)

▶ NVIDIA vk_micro_displacement (full)

Per-base-vertex bias & scale yields a ¼ 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.

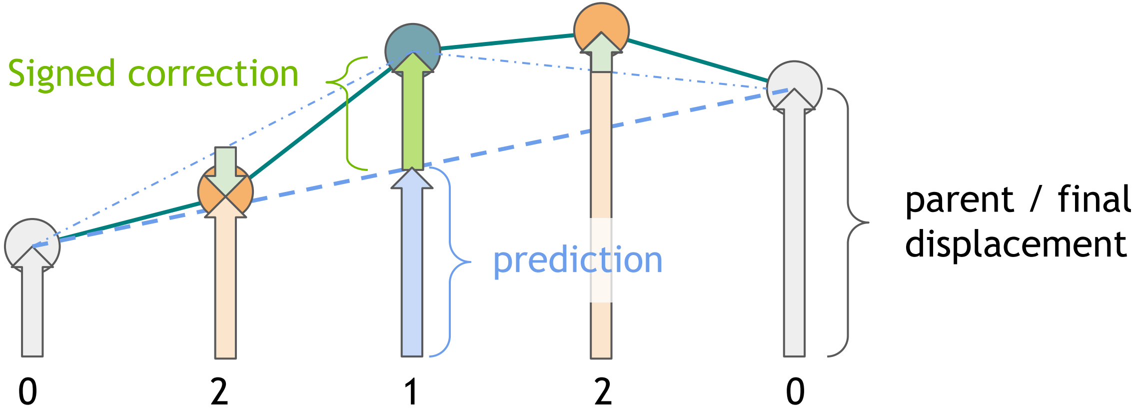
Displacement Compression

Displacement Compression

Hierarchical Delta Compression

Reconstruction of displacement:

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

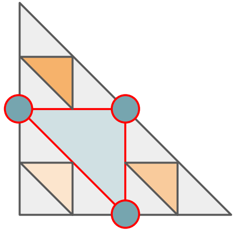


Microvertex Order

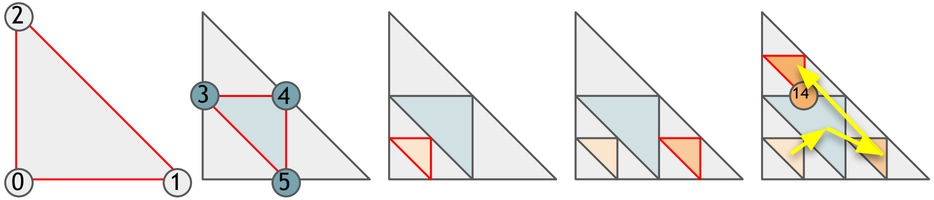
Example for subdivision level 2:
16 microtriangles, 15 microvertices

Vertex order within subdivided triangle

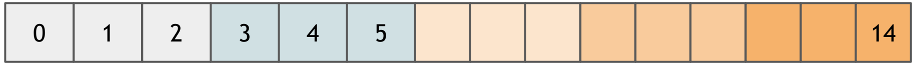
The “bird curve” order for microvertices is also based on hierarchical splitting, and stores vertices in groups of triplets.



Microvertex values are stored in triplets.



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



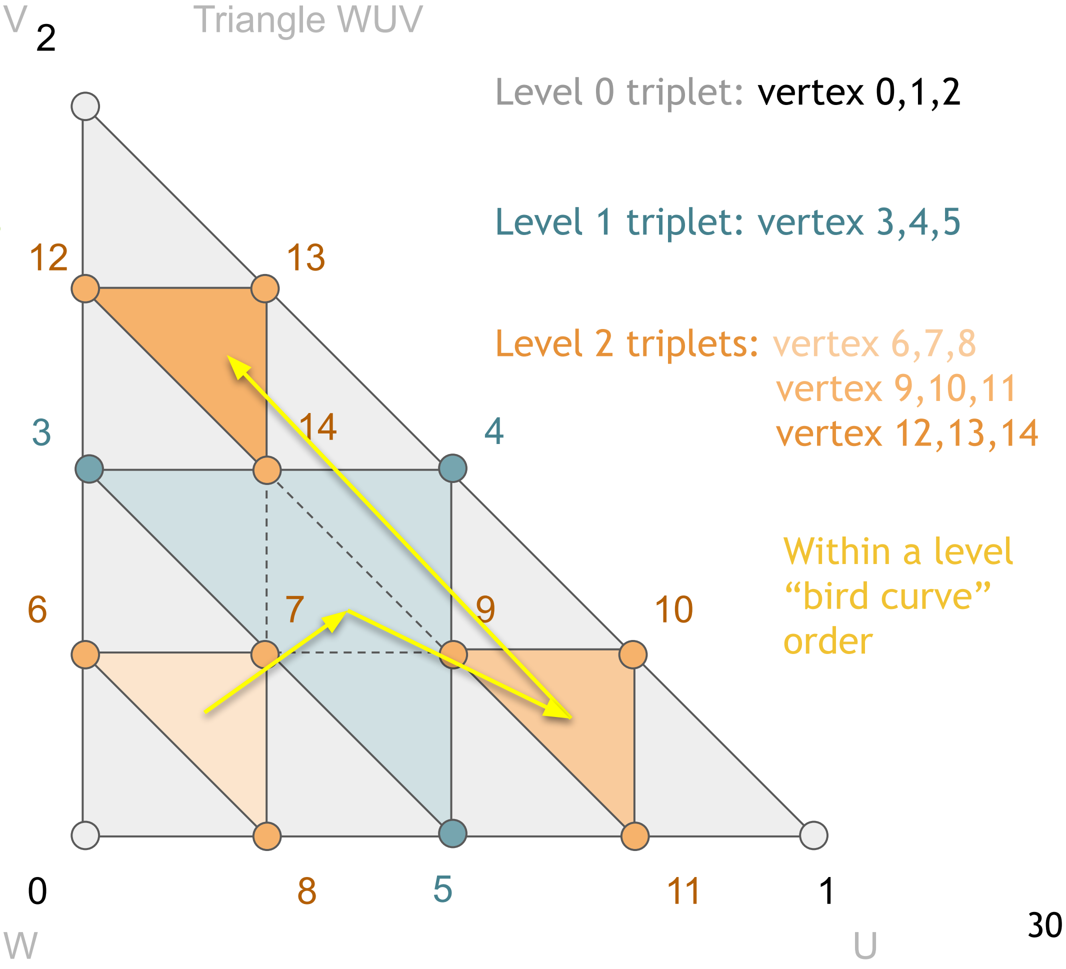
0	1	2	3	4	5									14

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.



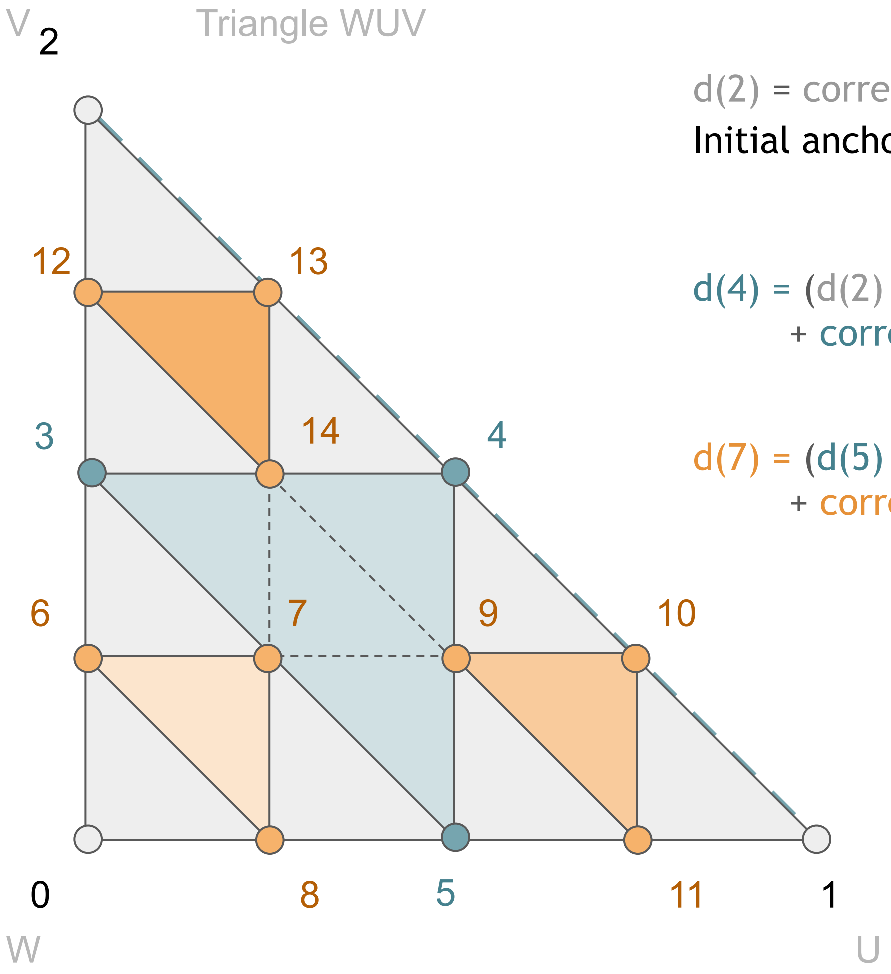
Compression

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.



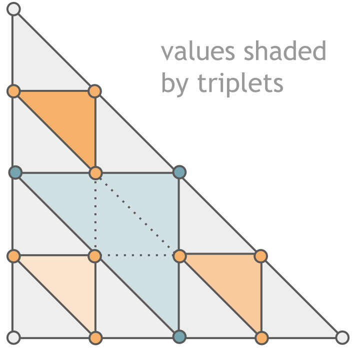
$$d(2) = \mathrm{correction}(2)$$

Initial anchors are lossless

$$d(4) = (d(2) + d(1) + 1) / 2 + \mathrm{correction}(4)$$

$$d(7) = (d(5) + d(3) + 1) / 2 + \mathrm{correction}(7)$$

Displacement Compression

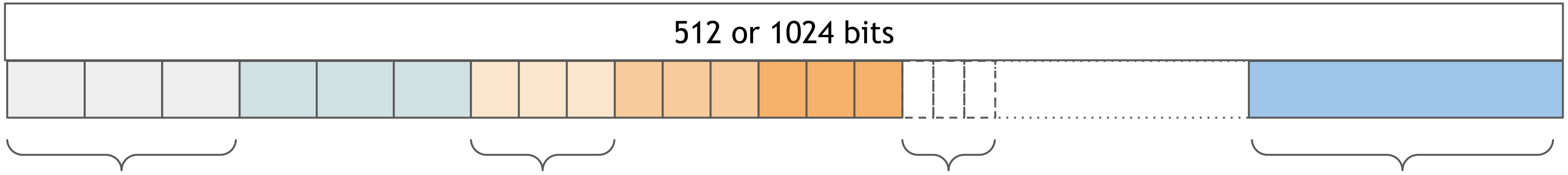


Block Layout

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)



Lossless
displacements first
(11 bits unsigned)

Displacement corrections
with fewer bits
(e.g. 8 bits signed)

(e.g. 4 bits signed)

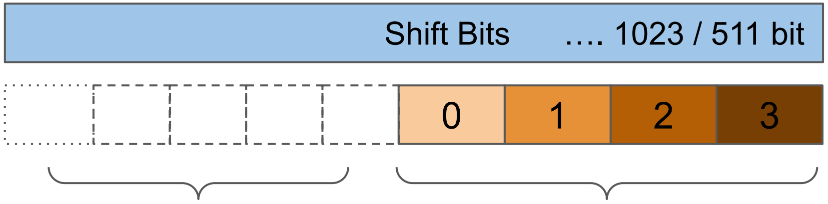
Shift bits for some
subdiv levels and
vertex types

Displacement Compression

Block Layout

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

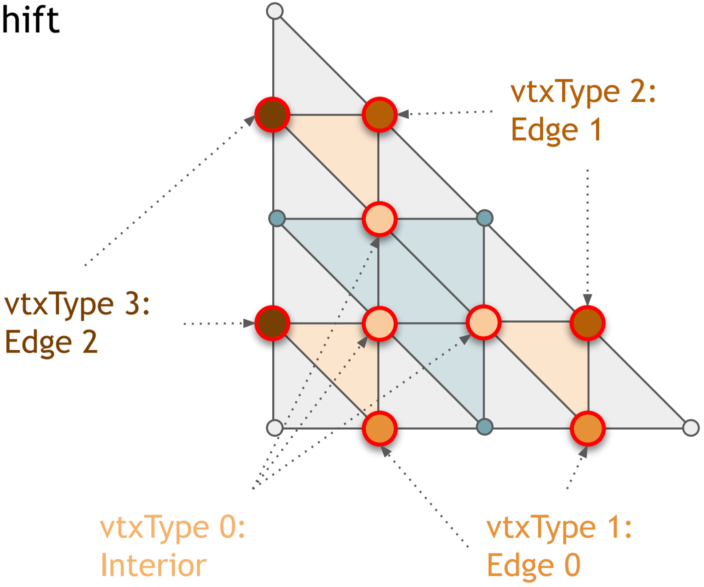
```
correction(vtx)
   = correction(vtx) << shiftBits[ vtxSubdivLevel ][ vtxType ];
```



Stored reversed, highest level in lower bits

Different shift values for each vertex type

Example for highlighted vertices with vtxSubdivLevel 2



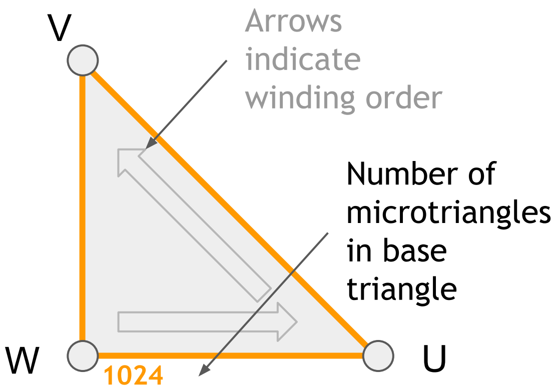
Displacement Compression

Illustration Guide

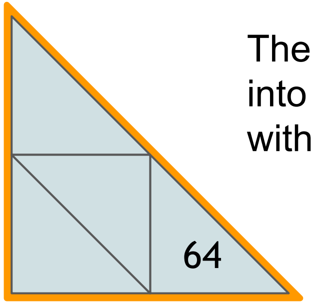
The next slides will use illustrations around base-triangles and sub-triangles.

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.



The base-triangle is defined from three vertices in order (W,U,V)



The base-triangle is split into 4 sub-triangles each with 64 microtris

Displacement Compression

Compression & Sub-Triangles

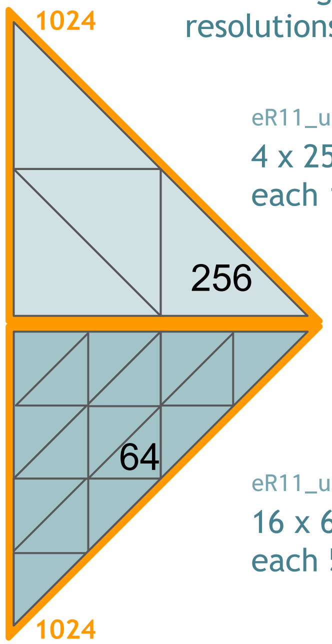
Similar to BCx compression we establish **block-compressed** formats that can represent displaced micro triangles at different fidelity in **512 or 1024 bits**. Decoded to **unorm11**.

```
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.

Example base-triangles each with 1024 microtris



Sub-triangles have equal resolutions

`eR11_unorm_lvl4_pack1024`
4 x 256 microtris blocks each 1024 bits

`eR11_unorm_lvl3_pack512`
16 x 64 microtris blocks each 512 bits

Displacement Compression

Compression Efficiency

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

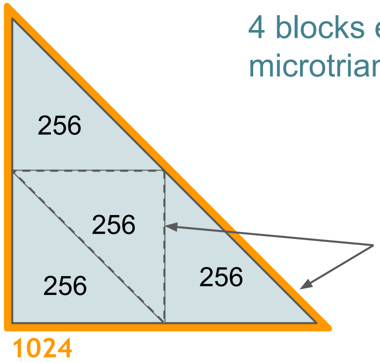
This can reduce overall compression a bit.

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.

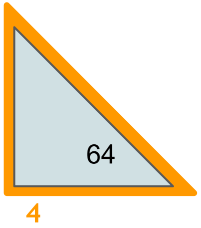
Base-triangle (1024 microtris)



4 blocks each 256 microtriangles

Displacement values along edges are stored redundantly in adjacent blocks (inner and outer edges)

Base-triangle (4 microtris)



Smallest blockformat offers 64 microtris, but base-triangle uses less, means we waste memory

Displacement Compression

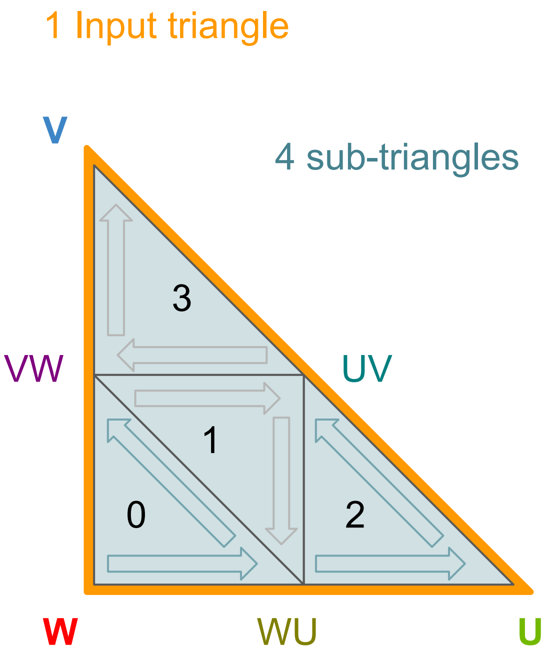
Compression & Sub-Triangles

Triangles are split using the “bird curve” rule seen before.

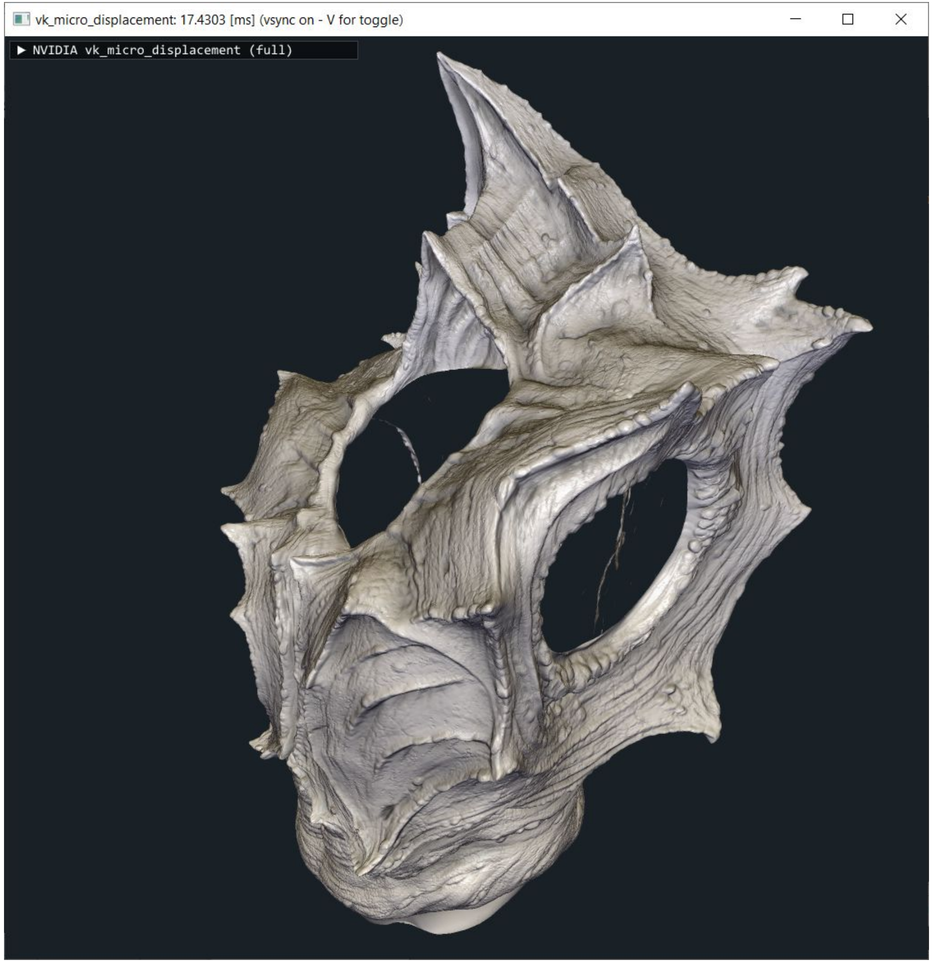
They are stored hierarchically in depth first order:

```
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



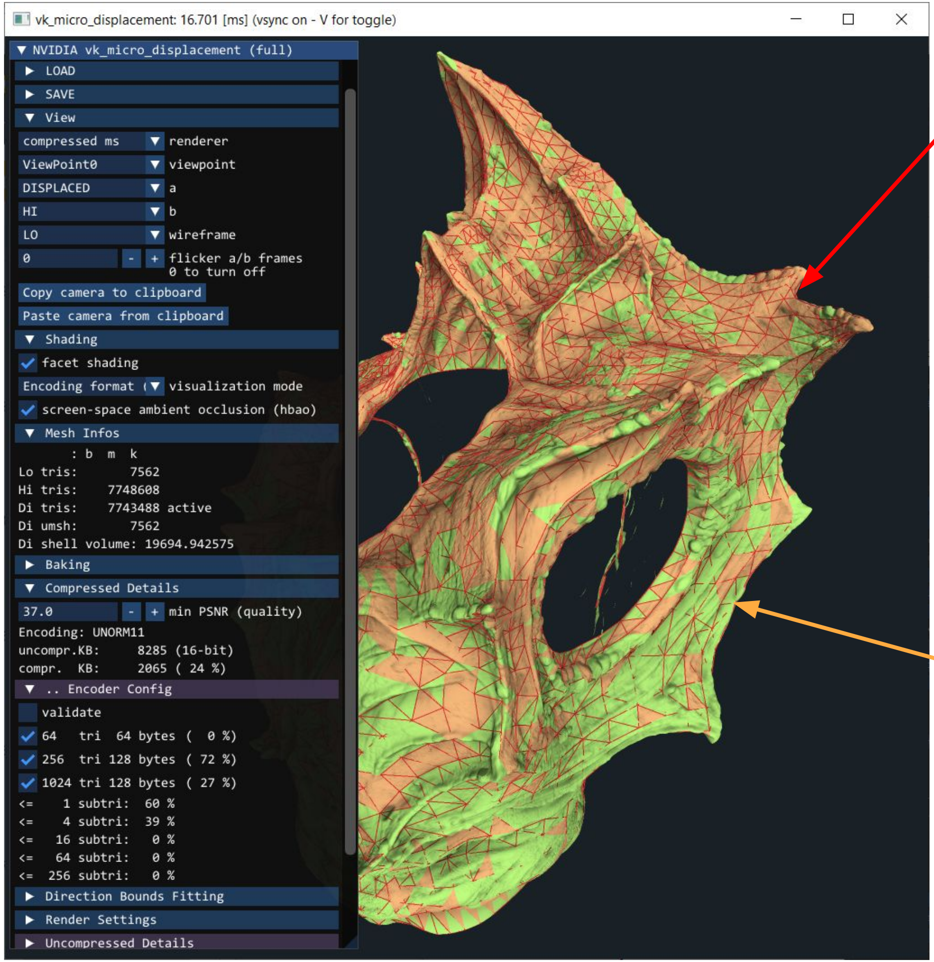
vk_micro_displacement: 17.4303 [ms] (vsync on - V for toggle)
▶ NVIDIA vk_micro_displacement (full)

Example

Facet-shaded and colored by sub-triangle

24 % compared to 16-bit uncompressed input

Different compression formats are used



```
vk_micro_displacement: 16.701 [ms] (vsync on - V for toggle)
▼ NVIDIA vk_micro_displacement (full)
  ▶ LOAD
  ▶ SAVE
  ▼ View
  compressed ms        ▼ renderer
  ViewPoint0           ▼ viewpoint
  DISPLACED            ▼ a
  HI                   ▼ b
  LO                   ▼ wireframe
  0                - + flicker a/b frames
                       0 to turn off
  Copy camera to clipboard
  Paste camera from clipboard
  ▼ Shading
  ☑ facet shading
  Encoding format (    ▼ visualization mode
  ☑ screen-space ambient occlusion (hbao)
  ▼ Mesh Infos
          : b   m   k
  Lo tris:        7562
  Hi tris:     7748608
  Di tris:     7743488 active
  Di umsh:        7562
  Di shell volume: 19694.942575
  ▶ Baking
  ▼ Compressed Details
  37.0             - + min PSNR (quality)
  Encoding: UNORM11
  uncompr.KB:      8285 (16-bit)
  compr.  KB:      2065 ( 24 %)
  ▼ .. Encoder Config
  ☐ validate
  ☑ 64   tri  64 bytes (  0 %)
  ☑ 256  tri 128 bytes ( 72 %)
  ☑ 1024 tri 128 bytes ( 27 %)
  <=    1 subtri:  60 %
  <=    4 subtri:  39 %
  <=   16 subtri:   0 %
  <=   64 subtri:   0 %
  <=  256 subtri:   0 %
  ▶ Direction Bounds Fitting
  ▶ Render Settings
  ▶ Uncompressed Details
```

Wireframe:

Lo-res base mesh

1024 x expansion:

7.5 K triangles to 7.7 M

Content-dependent compression:

White: 512bit per 64 mictrotriangles

Green: 1024bit per 256 microtriangles

Orange: 1024 bit per 1024 microtriangles

Additional Links & Information

These slides are part of the NVIDIA Micro-Mesh SDK
https://developer.nvidia.com/rtx/ray-tracing/micro-mesh

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