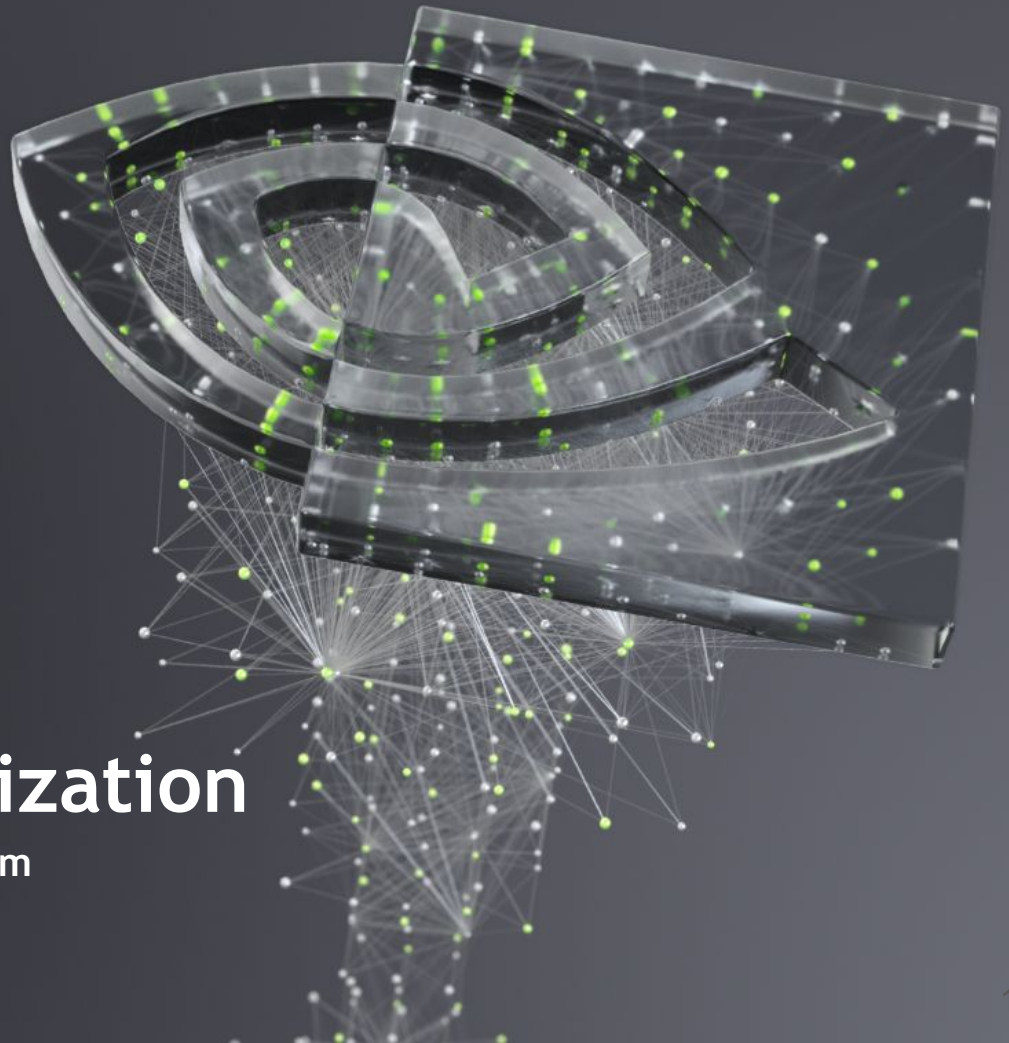


Micro-Mesh - Rasterization

Christoph Kubisch - ckubisch@nvidia.com

Note: Introduction to Micro-Meshes

Please refer to Micromesh Basics slide-deck first

This document focuses solely on the rasterization of displacement micromaps.

Micro-Mesh - Rasterization

Micro-Mesh Rasterization

New feature is primarily aimed at Ray Tracing hardware to reduce memory footprint.

Feasible to use current **Mesh Shaders** or **Compute Shaders** for rasterization

Displaced Micro-Meshes rasterization in games

- A lot of games use raster for primary visibility, this work allows matching rasterization with ray-tracing if desired. Or rendering displaced micromap assets without ray tracing.
- Works on various hardware in the market today

Displaced Micro-Meshes rasterization in professional space

- Sub pixel triangles are often a performance issue (wasteful geometry work if only a tiny fraction of triangles on sampling grid) → dynamic subdivision can help speed up rendering

This work also serves as a showcase how to do custom tessellation using mesh/compute-shaders beyond tessellation shaders

Micro-Mesh Rasterization Overview

Data Preparation

- Compute easy to look up values for each base-triangle
- Pre-compute reusable vertex and index tables for various subdivision / lod / edge-decimation permutations

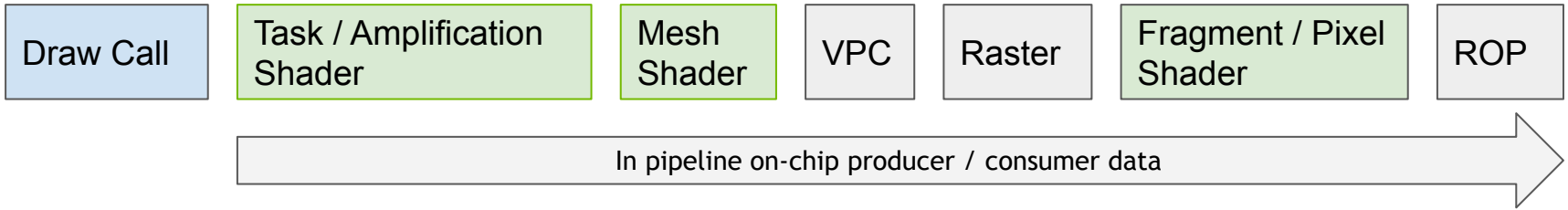
Task Shading

- Coarse Culling: Simple frustum and occlusion culling based on base-triangle bounding sphere.
- Dynamic Level of Detail: Adjust target subdivision level based on projected bounding sphere.
- Bin Packing: Optimize the output of meshlets to contain base-triangles or parts of base-triangles with equal subdivision level. Pack multiple low subdivision levels together.

Mesh Shading / Mesh Generation

- Triangle Topology: Based on target subdivision level and edge decimation setup triangle indices.
- Micro-Vertex Displacement Decoding: Fetch or decode the displacement values and compute the final transformed vertices.

Mesh Shader Pipeline



Task / Amplification Shader

- Each workgroup can generate up to 2M-1 mesh workgroups
- Each child workgroup has read access to task output payload

Mesh Shader

- Each workgroup can fill its pre-allocated meshlet (max 256 vertices, 256 triangles)

Optimal performance on current NVIDIA hardware in 2023

- Workgroup size matches one warp (32 threads)
- Meshlet size should not be max but ideally less (64 vertices, 84/126 triangles)
- Avoid shared memory usage, prefer subgroup/wave intrinsics and use NV_mesh_shader rather than EXT for more efficient primitive culling

Compute vs Mesh Shader

While the next slides mostly describe the mesh-shader solution. The compute solution works fairly similar. The decoder logic is the same, and the dynamic lod bin packing as well.

Compute-shaders also operate in units of 32 threads (warp/subgroup/wave) to make use of shuffle instructions (shared memory for decoding was slower), that is why the code-sharing is easy.

However, the workgroup size sometimes is increased to work on multiple jobs at once (a single warp workgroup can be inefficient for compute). As result, one compute-shader may do the same work as a task/mesh-shader, but we let multiple warps / independent jobs run in a single workgroup.

Further details about how the task/mesh-shader phases are mapped to compute are towards the end.

Triangle Splits for Rasterization

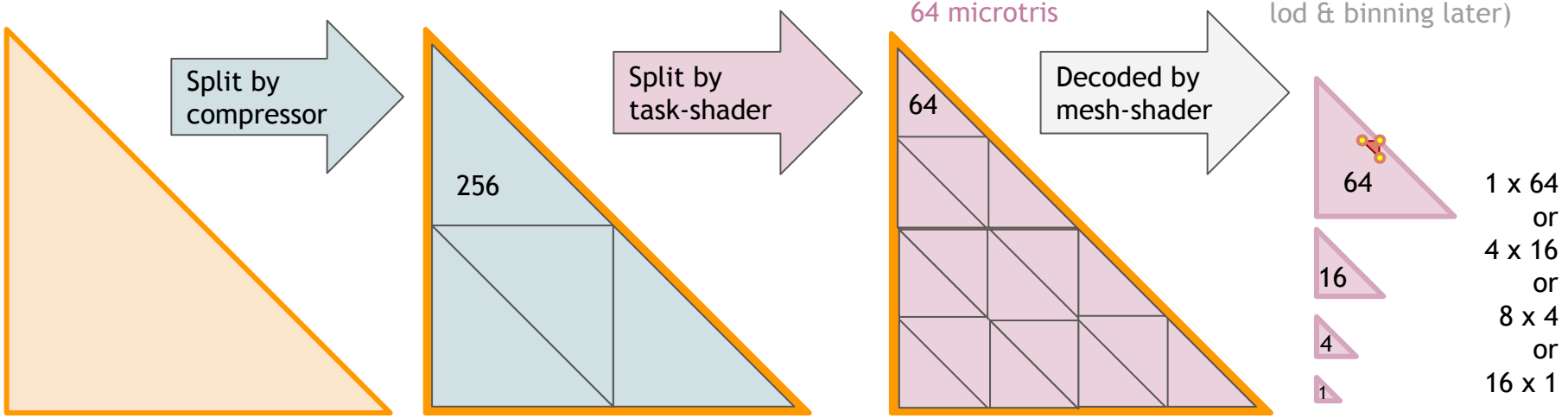
Base-triangle
(e.g. 1024 microtris)
== source content
artist / tooling

Sub-triangle
(e.g. 256 microtris)
== hw encoded block
512 or 1024 bits represent 64, 256 or 1024 microtris

Part-triangle
(up to 64 microtris)
== meshlet
mesh-shader decodes displacements for up to 60 microvertices & 64 microtris

Decode microvertices and setup microtriangles

Multiple parts can be worked on in one workgroup, see dynamic lod & binning later)



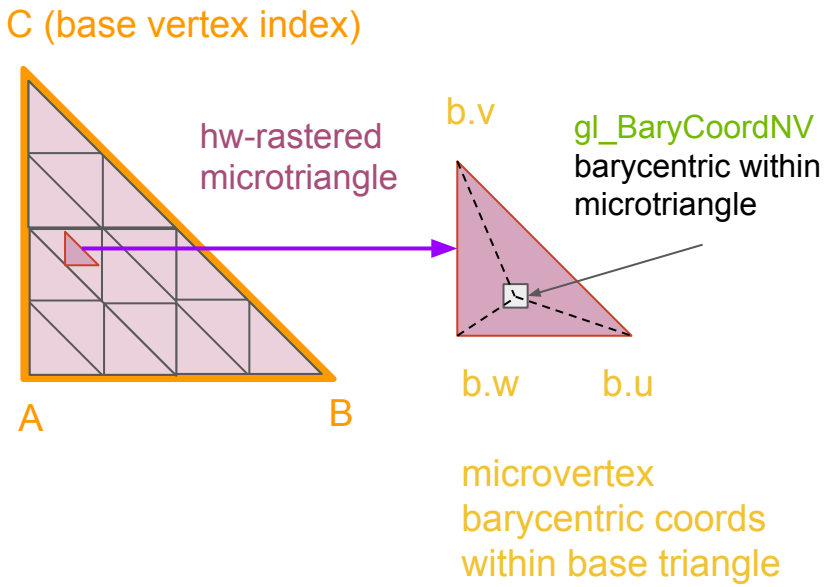
Rasterization Renderers

Common Optimization

- **Mesh Shader** doesn't output all interpolants (texcoords, tangents etc.) to save output space (improves occupancy)

- **Pixel/Fragment Shader** uses hw- interpolated micro-vertex barycentrics and base triangle vertex indices to compute shading attributes

  Can even use hw-barycentrics for per micro-vertex attribute interpolation



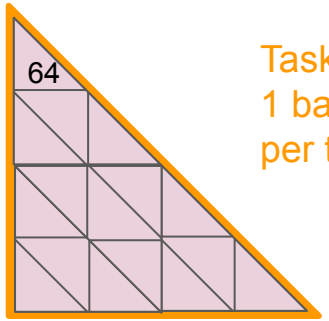
Rasterization Renderers

uncompressed renderers

- **Task shader** operates on
  1 base triangle per thread
  computes number of child-meshlets based on warp's base-triangles subdivision

- **Mesh shader** fetches pre-computed uv locations and indices of barycentric triangles and uncompressed data based on relative position within base-triangle

Base-triangle (e.g. 1024 microtris)
== source content



Task Shader:
1 base-triangle
per thread

Part-triangle
== meshlet
(up to 64 microtris)

Mesh shader:
64 vertices
64 triangles
32 threads

(1024 micro triangles
== 561 x 11 bits displacement data)

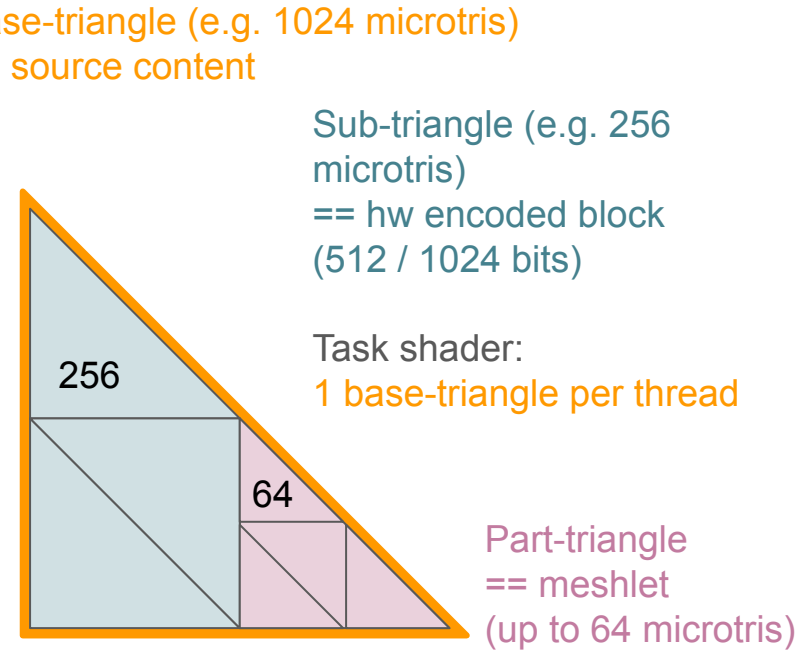
Rasterization Renderers

compressed renderers (block-compressed data)

- **Multiple implementations but similar principle**
- **Task shader:**
  1 base-triangle per thread.
  Computes number of child-meshlets based on warp's base-triangles subdivision

- **Mesh shader** also uses several pre-computed permutations for indices and uvs. Operates on sub- or base-triangle in full or partial.

  Single meshlet is limited to 64 triangles, but hw block may also represent more triangles, therefore additional splitting is done.



Displacement Decoder Implementations

Decoder Implementations

Micromeshes use block-compressed displacements, which leaves us two options to get displaced micro-vertices:

Intrinsic-based Decompression

There are intrinsics for mesh and compute shader that allow fetching micro-vertex attributes based on: `sceneTLAS, instanceID, geometryID, primitiveID, integer microVertexUV`

- Fetch float uv coordinates (edge decimation can cause micro-vertices to snap to different UV)
- Fetch object-space position

References directly the memory that the ray tracing scene uses, at reduced peak performance.

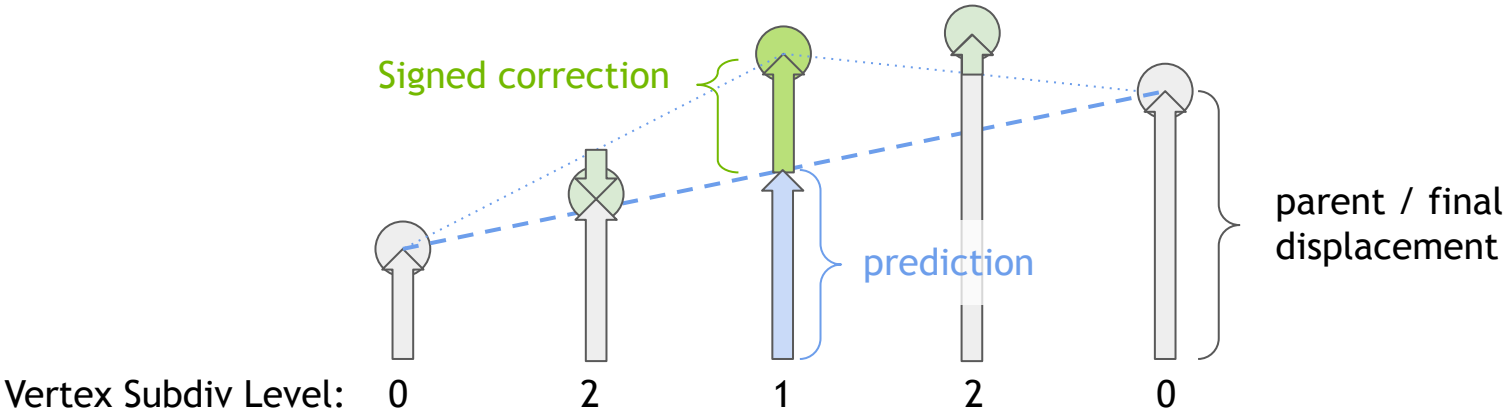
Manual Shader-based Decompression

The decompression is handled in the shader code by developers. There is sample code for this and following slides illustrate the process. Faster performance, but requires data next to ray tracing.

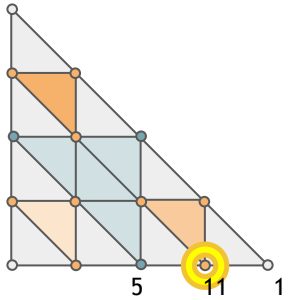
Manual Decoder Implementation

Principal Operations

- Fetch correction values based on compression format
- Get parent displacements, predict and apply signed correction if applicable.
  - Anchor verts and the entire 512-bit 64 micromap block are uncompressed unsigned.

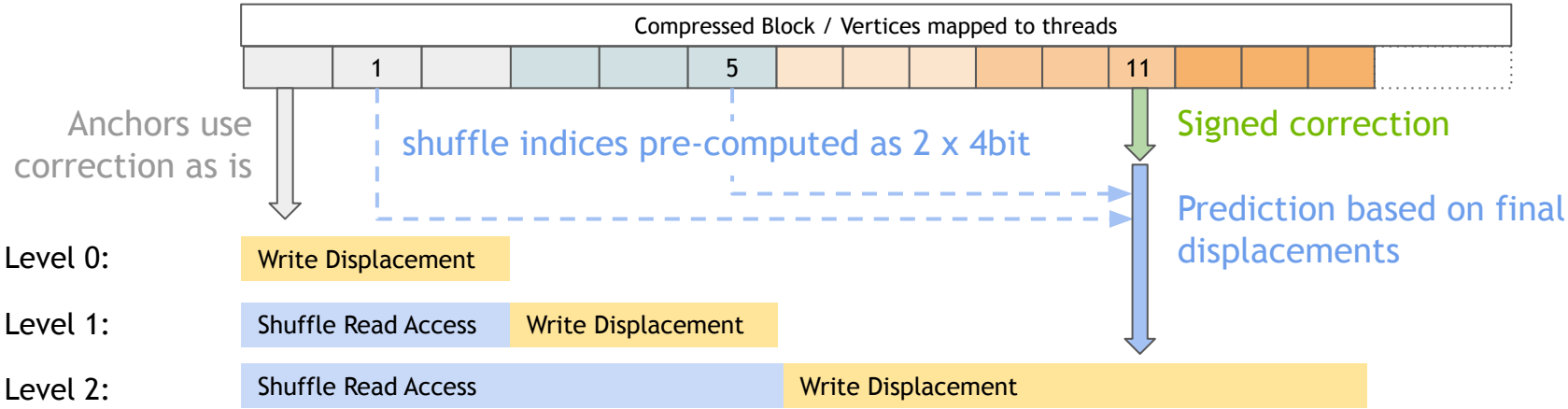


Manual Decoder Implementation



Gather-based Algorithm

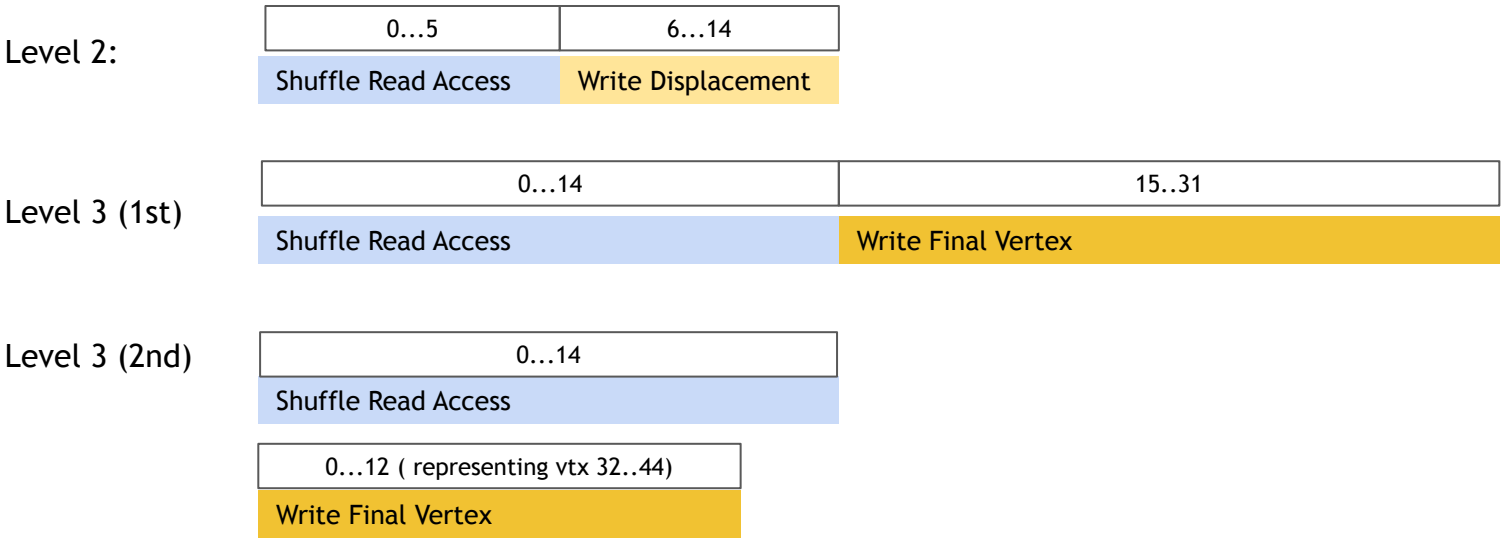
- Iterate subdiv levels, increase number of vertex threads, write into displacement register, but keep old threads active for read access
- Use shuffle to access decoded displacement registers of parents



Manual Decoder Implementation

Meshlet Caveats

- Meshlet limited to max 64 micro triangles / 45 vertices with 32 threads
- Use two iterations for 45 vertices, and directly write vertices

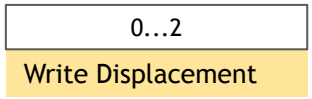


Manual Decoder Implementation

Meshlet Caveats

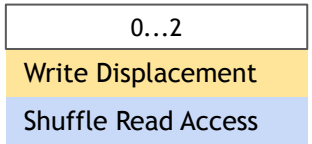
- Meshlet limited to max 64 micro triangles / 45 vertices (subdiv level 3)
- Sub-triangles/blocks with 256 or more microtris are split into multiple meshlets
- extra mip-block with uncompressed data (what the sample does) **or**
  descend hierarchy for local anchors, using pre-computed paths (illustrated below)

Level 0:



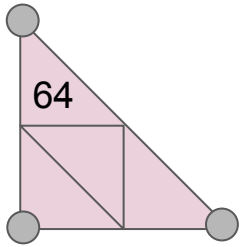
Original Anchors

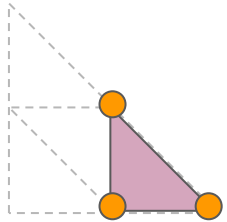
Level 1:



Adjusted local anchors

Level 2..4: Proceed as before, behaves like level 3





Example:
1 x Level 4 block with 256 triangles split into
4 x Level 3 meshlets with 64 triangles

Decoding and Mesh Generation

Initial Decode Phase

- 3 Threads = meshlet part-triangle: descend hierarchy or mip load initial anchor registers
- Iterate subdiv levels (excluding last, final vertex level),
  - N Threads = micro-vertex: computes displacement, saves in displacement register

Vertex Phase (requires 2 iterations: 32 threads for 45 vertices)

- V Threads = micro-vertex: compute last iteration displacement based on displacement registers and compute final vertex position.
  Also compute other vertex outputs (barycentric coord for shading etc.)

Primitive Phase (requires 2 iterations, 32 threads for 64 triangles)

- P Threads = micro-triangle: fetch pre-computed indices based on LoD level and relevant edge-decimation permutation, adjust primitive indices / winding.

Decoder Implementations

Multiple decoders exist for the compressed renderers

For research purposes multiple decoders for the compressed data were implemented during prototyping.

The next table shows those available in the open-source sample, as they gave best results.

Decoder Implementations

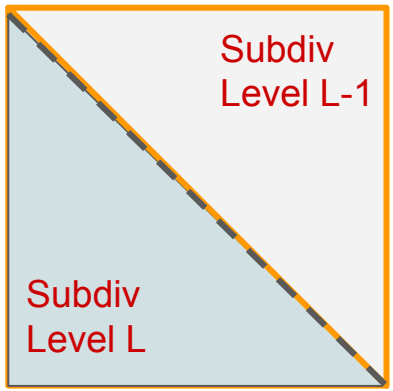
compressed rasterization renderers (block-compressed data)

Decoder	**Input Frequency**	**Auxiliar Data** (+ 128-bit for precomputed cull/lod sphere is common)	**Decoding Logic**	**Performance**	**Re-use Raytracing Data Directly**
Base-Triangle w. Mip	base-triangles	64-bit per input **AND** 192-bit mip-block for each base-triangle that uses 256 or 1024 microtris blockformats	Gather-based decode via shuffle across subgroup	++	-
Micro-Triangle	base-triangles	64-bit per input	Decodes micro-triangle per-thread, picks micro-vertex from micro-triangle corner	--	-
Micro-Triangle Intrinsic	base-triangles	64-bit per input	Fetches micro-vertex through intrinsic	o	x

Base Triangle Properties

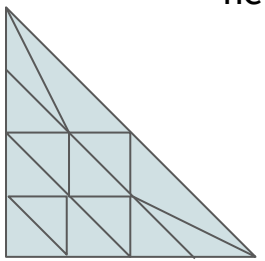
Common inputs

- All inputs are always flattened so no indirection is used
- Base triangles are allowed 1 level of subdivision difference
- Stored into header to avoid indirections
- micromesh topology encodes local permutation for the watertightness handling



	bits
base topology	3

Per edge set bit if half-resolution neighbor exists



Topology value (0..7) encodes index buffer permutation

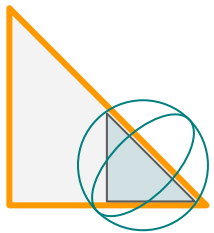
Base Triangle Properties

Dynamic lod inputs

- Bounding sphere helps frustum culling and LoD computation
- max displacement in primary header is meant for animated content where sphere is computed on-the-fly
- Extra 128-bit stores pre-computed sphere (no special fitting, just triangle center) for static content. Avoids indirections.

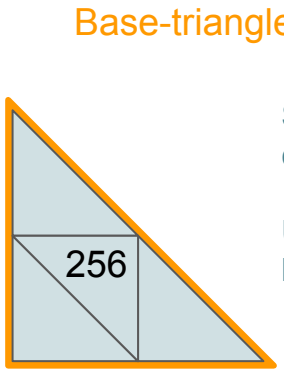
	bits
max displacement	8

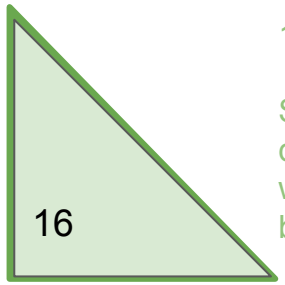
	bits
sphere position	96
sphere radius	32



Base-Triangle w. Mip Decoder

- Inputs are base-triangles, works very similar to the “uncompressed”, pre-computes meshlets for different LoD levels and compression formats
- **Task shader** 1 base-triangle per thread
- **Mesh shader** operates on parts of 64 triangles (or less)
- Dynamic-lod within base-triangle
- Shuffle and pre-computed Mip-triangle to decode displacements from blocks
- Can get close to uncompressed renderer.

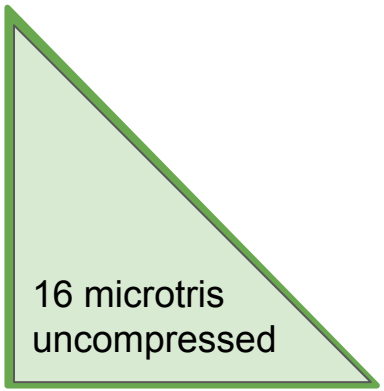


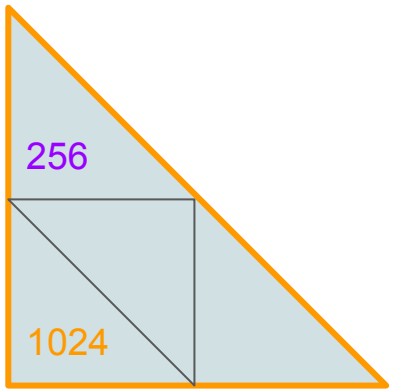


Base-Triangle w. Mip Decoder

Input

- 64-bit information per base-triangle
- All sub-triangles have same format / subdivision
- 192-bit mip-triangle, **only** if base-triangle uses 256 or 1024 microtris blocks



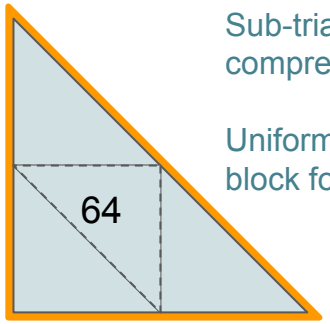


	bits
base subdiv level	3
block format	2
base topology	3
max displacement	8
mip data offset	22
compressed data offset	26
TOTAL	64

Micro-Triangle Decoder

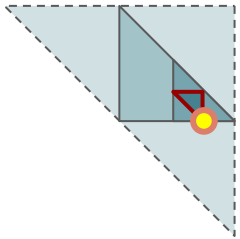
- Inputs are base-triangles, works very similar to the “uncompressed”, pre-computes meshlets for different LoD levels and compression formats
- **Task shader** 1 base-triangle per thread
- **Mesh shader** operates on parts of 64 triangles (or less)
- Dynamic-lod within base-triangle
- Brute-force decodes microvertex per-thread
  - Use pre-computed table to find which microtris and sub-triangle block it belongs to
  - Decode microtri by descending hierarchy
  - Microvertex is one corner of microtri
- Descending is **very slow** for both implemented versions:
  - Pre-computed decoding path (faster)
  - Full ALU-based solution (slower)

Base-triangle (e.g 256)



Sub-triangle == compressed block

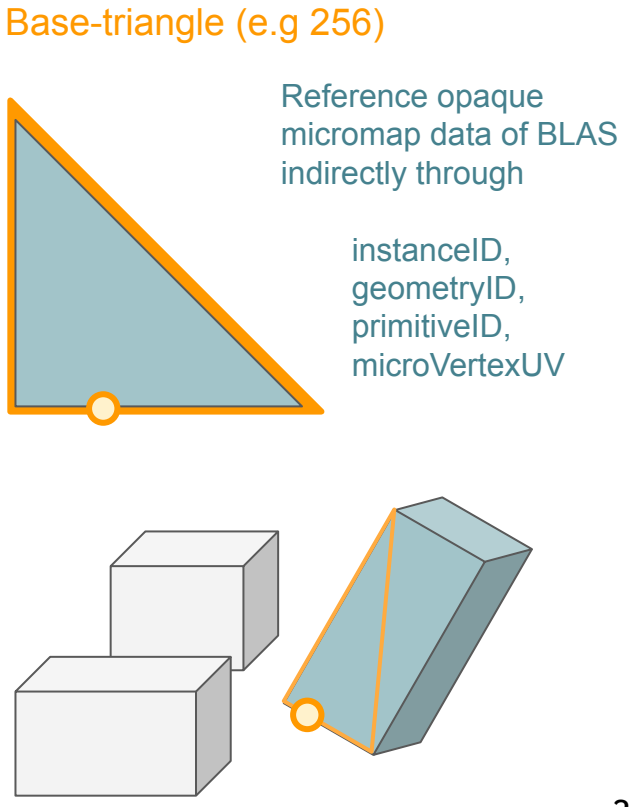
Uniformly split using same block format.



Descend triangle decoding within block to target micro-triangle
Fetch microvertex displacement from it

Micro-Triangle Intrinsic Decoder

- Inputs are base-triangles, works very similar to the “uncompressed”, pre-computes meshlets for different LoD levels and compression formats
- **Task shader** 1 base-triangle per thread
- **Mesh shader** operates on parts of 64 triangles (or less)
- Dynamic-lod within base-triangle
- Fetches microvertex per-thread
  - Use pre-computed table to get micro vertex UV
  - Fetch object-space position using intrinsic



Rasterization Inputs Overview

compressed renderer inputs

Data-independent

- Pre-computed vertices (uv and decompression info)
- Triangle indexbuffers for various topology / edge decimation permutations (heavily re-used)
- Other information that aids decoding (descending paths, etc.)
- Several of these tables are pre-computed for multiple subdivision level, dynamic lod and block format permutations

Rasterization Inputs Overview

compressed renderer inputs

Mesh-dependent

- Base mesh index & vertex buffer
- Base mesh direction buffer (fp16)

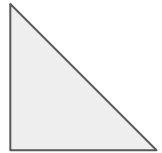
Displacement-dependent

- (Base mesh direction bounds buffer (fp16 or fp32), optional)

- Compressed displacement buffer (u32)
- 64-bit per base-triangle
- 128-bit per base-triangle LoD pre-computed sphere buffer
- (192-bit mip-triangle for some base-triangles for the “base w. MIP” decoder)

Dynamic Level of Detail and Bin Packing

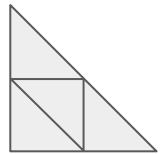
Dynamic LoD



Micro-Meshes subdivision scheme allows by design dynamic level of detail

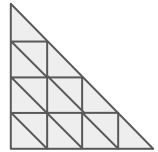
Watertight representations are crucial in high quality rendering.

To avoid cracks, dynamic LoD can be adjusted with a bias limit of up to 3 levels on a per-instance level. Further levels could break watertightness within a micromesh.



Raytracing

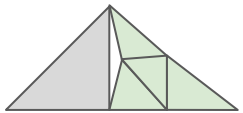
Does **not** expose dynamic LoD control



Rasterization

The use of LoD bias is up to the developer's implementation, though it can create a mismatch compared to the raytracer.

For rasterization performance and visual quality it is beneficial to make LoD bias decisions unlimited and per base-triangle, rather than per instance. LoD transitions in sub-pixel space and anti-aliasing techniques can hide watertightness issues. Alternatively more sophisticated per-edge LoD schemes would be required.

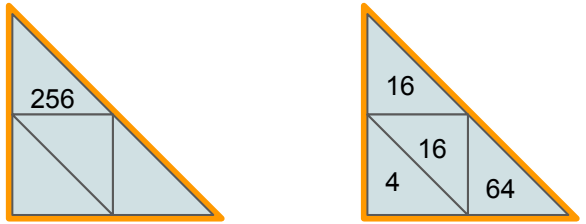


Dynamic LoD

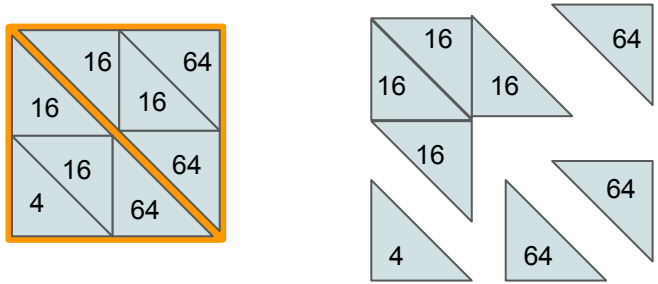
Renderer with dynamic lod

- Compute LoD subdiv per base-triangle (projected sphere size to drive dynamic LoD)
- Currently not watertight / no per-edge LoD (getting away with it, being mostly sub-pixel)
- No backface-cluster cull in sample (may benefit hardsurface models/CAD)

- Task Shader bins & packs multiple low subdiv sub-triangle/blocks into a single meshlet
- Mesh shader unpacks and may decode entire, partial or multiple blocks at once



Original Subdiv → Dynamic Subdiv



Batch multiple blocks in single mesh shader invocation (e.g. total of five here)

Dynamic LoD

Task shader needs to cull, bin & pack base-triangles

- Bin by same effective subdivision level (0..3) (higher levels are rendered as multiple part-triangles with 64 microtris) into meshlets.
- The packing should be tight, so that mesh shader makes best use of the meshlet space it allocates
- Ideally minimize task shader output space

Mesh shader may need do decode multiple sub- or base-triangles within same warp

- Unpack the decoder state from task shaders output.
- For best packing efficiency must handle level > 3 next to regular level <= 3 triangles
- Effectively less threads and no longer fully uniform code-path

Bin Packing

Packing Configurations

- Multiple input triangles of same subdiv level fit in single meshlet
  (may under-utilize output space a bit)
- Decoder state is same for group of threads working on same input triangle
  (mesh shader is limited to single warp in total, compute shader wouldn't, but didn't benefit)

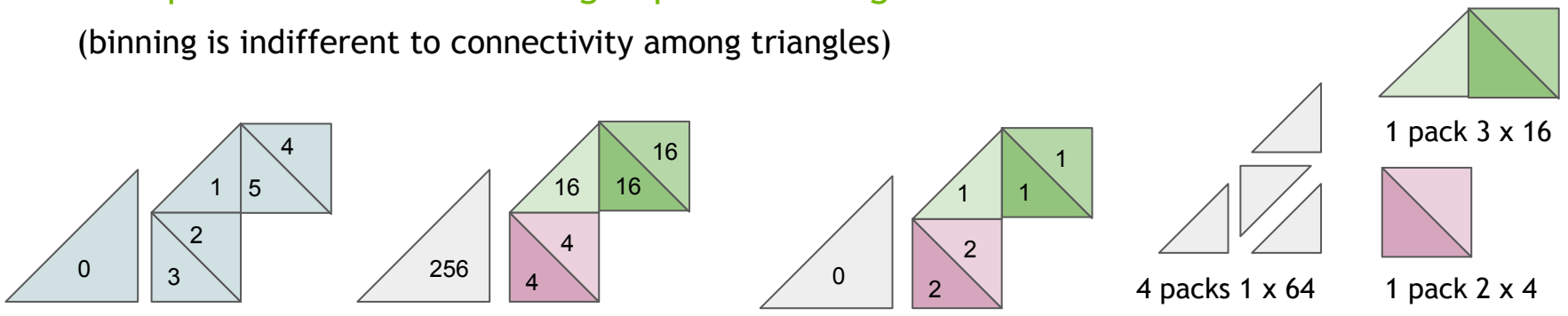
subd level	vertices per micromesh	triangles per micromesh	packed micromeshes	threads per micromesh	total vertices	total triangles
0	3	1	16	2	48	16
1	6	4	8	4	48	32
2	15	16	4	8	60	64
3	45	64	1	32	45	64

Mesh shader packing configurations
(always 64 v, 64 t, 32 threads)

Bin Packing Example

Example for task shader workgroup of six triangles

(binning is indifferent to connectivity among triangles)



Base-triangle IDs

0	1	2	3	4	5

Target micro triangles

256	16	4	4	16	16

Target bins = total 3

0	1	2	2	1	1

Bin meshlets = total 4 + 1 + 1

4	1	1	1	1	1

Task shader output is tightly packed by bin and ID

0	1	4	5	2	3

Task output IDs

Bin Packing

Task shader output

- Common start ID
- Per output triangle there is various information about the block and the bin/packing configuration

Reminder: One mesh-shading workgroup can process only up to subdiv 3 at a time. Need multiple workgroups for more.

The sample used a few more bits than shown here and as result 32 bit per triangle + 32 bit base ID.

output	bits	count
start triangle ID	32	1
relative triangle ID	5 (fits 0..31)	32
target subdiv level	3 (fits 0..5)	32
bin.meshlets.start (first linear meshlet index of this bin)	9 (fits 31 x 16)	32
bin.pack size	2 (represents 1,4,8,16)	32
bin.offset (where in output bin starts)	5 (fits 0..31)	32
TOTAL	**32 + 8 x 32 + 16 x 32 = 100 bytes**	

Bin Packing

Task shader micromesh threads

	0	1	2	3	4	5
target subdiv	4 (as 3)	2	1	1	2: 16 tris	2
subdiv match	100000	010011	001100	001100	010011	010011
bin.size	1	3	2	2	3	3

Task shader bins 32 micromeshes

- Computes target subdiv level or culls
- Uses `subgroupPartitionNV` instruction to match and bin meshes of same target subdiv

Bin Packing

Task shader bins 32 micromeshes

- Computes target subdiv level or culls
- Uses `subgroupPartitionNV` instruction to match and bin meshes of same target subdiv

- Computes number of meshlets each bin needs and can pack base-triangles into single meshlet
  (1,4,8,16 x base-triangles)

Task shader micromesh threads

	0	1	2	3	4	5
target subdiv	4 (as 3)	2	1	1	2: 16 tris	2
subdiv match	100000	010011	001100	001100	010011	010011
bin.size	1	3	2	2	3	3
bin.pack size	1	4	8	8	4	4
bin.meshlets	4 (4 x subiv 3)	1	1	1	1	1
bin.msh.start	0	4	5	5	4	4

Bin Packing

Task shader bins 32 micromeshes

- Computes target subdiv level or culls
- Uses `subgroupPartitionNV` instruction to match and bin meshes of same target subdiv

- Computes number of meshlets each bin needs and can pack base-triangles into single meshlet
  (1,4,8,16 x base-triangles)

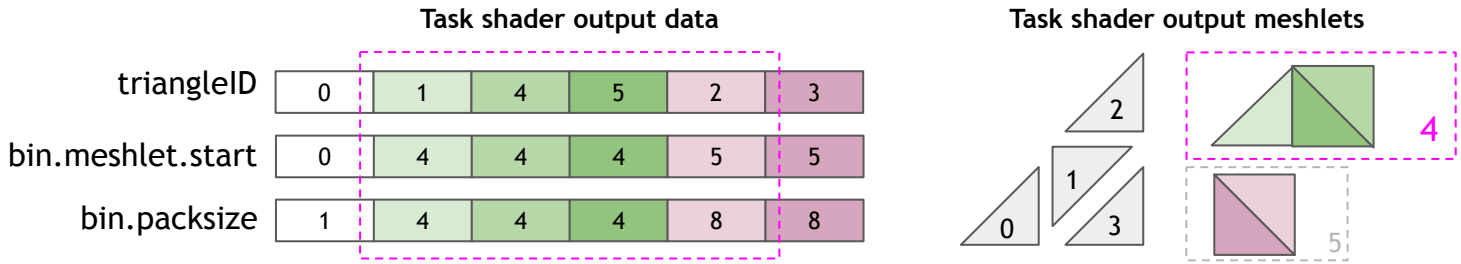
- Computes bin offsets and writes out triangle infos grouped by bins

Task shader micromesh threads

	0	1	2	3	4	5
target subdiv	4 (as 3)	2	1	1	2: 16 tris	2
subdiv match	100000	010011	001100	001100	010011	010011
bin.size	1	3	2	2	3	3
bin.pack size	1	4	8	8	4	4
bin.meshlets	4 (4 x subiv 3)	1	1	1	1	1
bin.msh.start	0	4	5	5	4	4
bin.offset	0	1	4	4	1	1
match.offset	0	0	0	1	1	2
out.offset	0	1	4	5	2	3

task output. triangle relative ID	0	1	4	5	2	3

Bin Unpacking Example



Unpacking happens in the Mesh Shader.

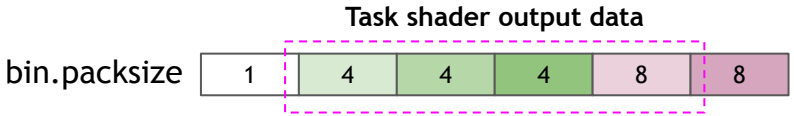
The Task shader emitted **6** Mesh Shader workgroups (0...5) in total.

The Mesh shader workgroup index for the highlighted meshlet is 4 and packs up to 4 x 16 microtriangles.

At the start of mesh shader we need to figure out where its bin starts in the task output data.

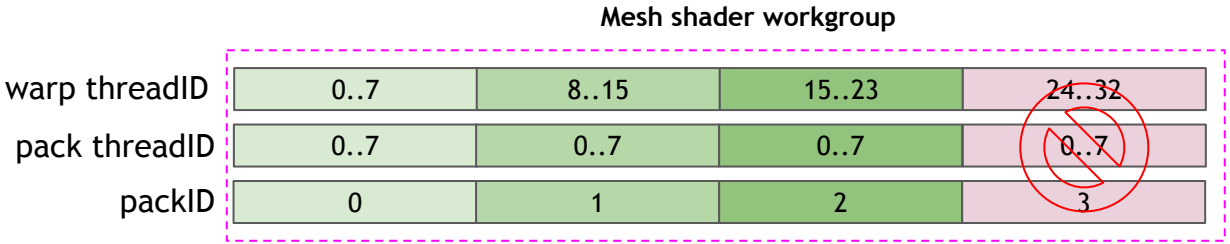
The **bin.meshlet.start** values are ascending (prefix sum). We load them across the mesh shader warp and each thread runs a comparison (**meshletIndex >= bin.meshlet.start**). With subgroup/wave intrinsics we broadcast the first winner thread index, which then is equivalent to our start index into the output data (here 1).

Bin Unpacking Example



Divide the mesh shader warp in packsize many regions, each operates on one micromesh.

Discard micromeshes that differ from first thread's packsize or exceed task output.



Use "pack threadIDs" instead of regular thread IDs. Rest of decoding works the same.

We still use two iterations for the vertices and primitives.

In this example we have 15 vertices per micromesh but only 8 threads. Shuffle access still works, because previous subdiv level required 6 vertices/threads, though shuffle indices need to be adjusted for the thread region.

Bin Unpacking

Determine task output micromeshes to work on in mesh shader warp based on meshlet index

Mesh shader computes triangles & vertices

- Figures out bin config (pack size) and active micromesh ids

Bin Unpacking

Mesh shader computes triangles & vertices

- Figures out bin config (pack size) and active micromesh ids

- Each micromesh uses even distribution of threads within warp
- Setup per micromesh decoder configs

Determine task output micromeshes to work on in mesh shader warp based on meshlet index

Distribute micromesh threads evenly across warp

	0..7	8..15	16..23	24..32
micromesh relative ID	1	4	5	2
pack size	4	4	4	1
subdiv	2	2	2	3
valid: pack size == first	true	true	true	false

Bin Unpacking

Mesh shader computes triangles & vertices

- Figures out bin config (pack size) and active micromesh ids
- Each micromesh uses even distribution of threads within warp
- Setup per micromesh decoder configs
- Two warp iterations to process all micro vertices / triangles
  (e.g. single triangle: 16 x 2 threads, 3 vertices max)

Determine task output micromeshes to work on in mesh shader warp based on meshlet index

Distribute micromesh threads evenly across warp

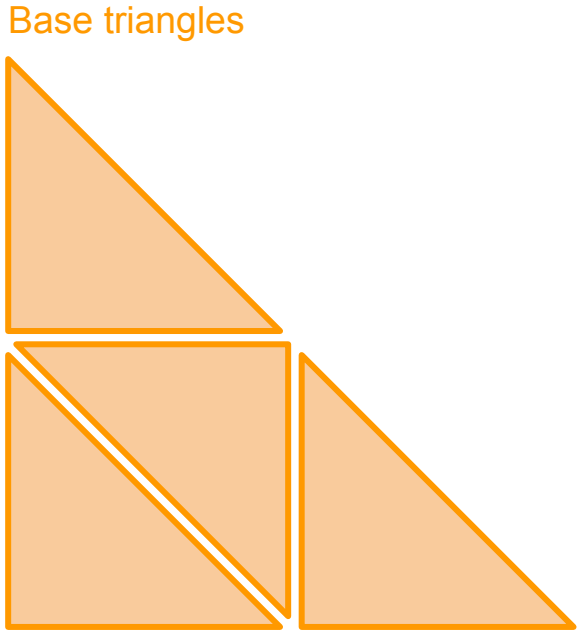
	0..7	8..15	16..23	24..32
micromesh relative ID	1	4	5	2
pack size	4	4	4	1
subdiv	2	2	2	3
valid: pack size == first	true	true	true	false

1st vertices / tris	0..7	0..7	0..7	0..7
2nd ...	7..15	7..15	7..15	7..15

Dynamic LoD Caveats

Open Issues

- Renderer looks at base-triangles individually, therefore no “vertex re-use” for connected base-triangles when they are all subdiv 0. One should switch to traditional pre-computed meshlets in this case.
- To get deeper ranging LoD for minification, multiple traditional LoD models for the same object could be used and swapped out, or a hierarchical cluster LoD scheme that operates at finer granularity across the object.



Dynamic LoD Outlook

Potential Improvements

- When base-triangles' dynamic lod predominantly is level 0 (i.e. one output triangle per base-triangle) should render model with traditional index-buffers or via meshlets, both improve vertex re-use among connected base-triangles
- No research yet for models with multiple LoD models spanning different detail levels
  - LoD Model 0: dynamic reduction of up to 1024 triangles
  - LoD Model 1: another dynamic reduction… could use hierarchical LoD etc.

Level of Detail

- Dynamic LoD can greatly improve performance and mostly hide transitions subpixel
- Current implementation is not water-tight, simple logic
- Still may need multiple LoD models / schemes for background minification

Compute Shader Rasterization

Compute Shader Rasterization

The compute shader rasterization in this sample is rather basic and not specifically tuned. Vertices are transformed across the warp and stored in shared memory. Each thread then operates on one triangle:

- Pulls the relevant vertices
- Compute the screen-space rectangle for visible triangles
- Loop over rectangle pixels, test the sampling point against the triangle
- If the triangle is covered, write out pixel via atomic 64-bit min operation, where upper bits store depth and lower bits the payload

Given we mostly have smaller triangle sizes these simple loops work good enough.

When triangles would require clipping, they are currently discarded. The better approach is to leave more complex triangles to hardware rasterization.

Compute Shader Rasterization

DUAL PASS: First pass does task-shading, then second pass does mesh-shading phase

Task shading pass

- Handles LoD bin packing and culling, writes to global memory
- ! Needs upper-bound of visible bins for scratch buffer (12 bytes per visible meshlet)

Mesh shading pass

- DispatchIndirect based on task phase results
- Same single warp decoders as mesh-shader
- Rasterize micro triangle via atomic 64 bit (payload & depth) directly per thread.
- May want to store barycentric coordinate via second 64-bit atomic, to avoid recalculation, at risk of some errors/contention on identical z values and extra memory.

Pixel shading pass

- Unpack payload information per-pixel for gbuffer fill
- The sample doesn’t implement any proper shading, so the payload usage is not realistic.

Compute Shader Rasterization

DUAL PASS SPLIT: task-shading pass can also rasterize smaller bins immediately, then second pass to rasterize larger subdivisions in mesh-shading phase

Task shading pass

- Handles LoD bin packing and culling,
- Splits bins depending on local subdivision level
  - Writes **big** (subdiv == 3) bins to global memory
  - Rasterizes smaller packed bins directly, these handle multiple basetriangles at once
- ! Needs upper-bound of visible bins for scratch buffer (12 bytes per visible **big** meshlet)
  This bound is typically lower than the previous technique, as we don't need to output those smaller bins.

Mesh shading pass

- (same decoding and rasterization as before)

Pixel shading pass

- Unpack payload information per-pixel for gbuffer fill

Compute Shader Rasterization

SINGLE PASS (slower): Collapse task and mesh-shading into single workgroup.

(was removed from sample due to slower performance)

Combined pass:

Task shading phase (outputs to shared memory)

Mesh shading phase

- Workgroup uses 4 warps (128 threads) and therefore needs to loop over the total amount of work that the task shading phase generated. Where a task-shader spawned N mesh-shader warps, compute distributes these across 4 within workgroup and loops.
- (otherwise similar decoding as previous slides)

Pixel shading pass

- Unpack payload information per-pixel for gbuffer fill

Basic Performance Tests

Performance Test Description

The shading is very basic phong, no textures and we measure two variants

- Shaded with micro-vertex normals
  - This looks a bit nicer than just flat-shading, it does add a bit of extra work
- Flat-shaded colored triangle IDs (to mimic a visibility buffer like scenario)
  - This is the only variant supported by compute-shader rasterization in this sample

We do not compare to rendering the hi-res model by traditional means as it is typically slower and would require more sophisticated cluster schemes, vertex quantization etc. to be sped up.

However, in a scenario where low or no subdivision is applied, those techniques would be better, as was discussed earlier for dynamic LoD.

Performance Test Description

Murex Romosus from Three D Scans

Mesh Stats:

Basemesh: 19 K Triangles

Displaced mesh: 3.8 M Triangles (<= subdiv level 4 average)

144 Instances ~ 540 M Triangles

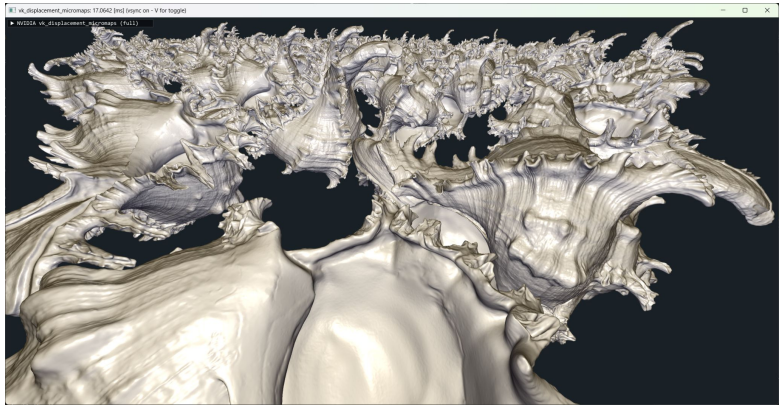
Micromap Data:

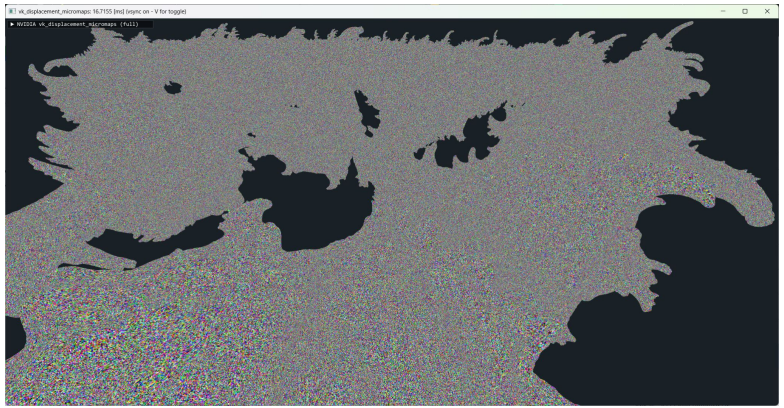
Uncompressed displacement (densely packed 11 bit): ~ 3.1 MB

Block compressed displacement: ~ 2.4 MB

Block compressed displacement w. mips: ~ 2.7 MB

32 bit octant microvertex normals: ~ 9 MB





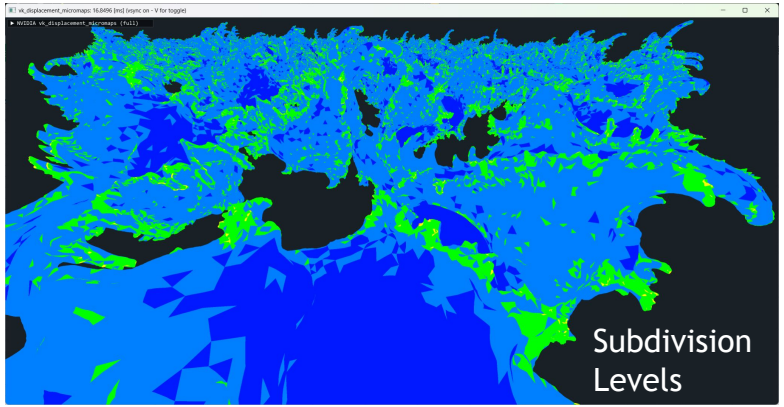
Micro-Mesh model from Micro-Mesh Construction - Example of results (unimi.it)

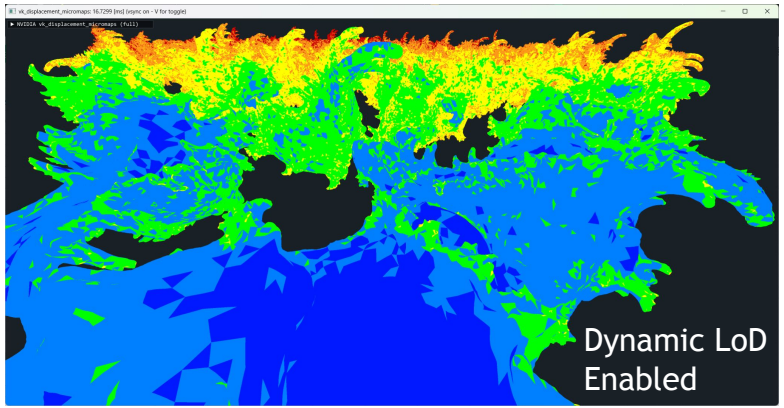
Performance Test Description

Both images show the base-triangle subdivision level.
Level 0 is red, Level 5 is dark blue.

Dynamic LoD is activated in the lower image and significantly reduces number of rendered triangles.

No LoD:	540 M triangles
LoD:	25 M triangles
LoD + occlusion culling:	15 M triangles





Performance Test Description

Occlusion Culling

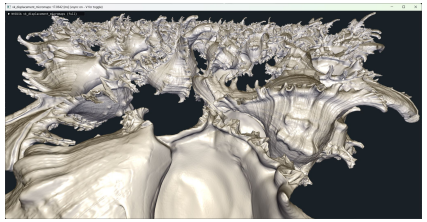
Very simple occlusion culling was implemented. It is only correct on a static frame as it uses the previous frame depth buffer. A HiZ mipmap chain is built and the bounding spheres of displaced base-triangle is tested against it. This hasn’t been tuned much yet. For example ideally some coarser level occlusion culling should be done on groups of base-triangles.

Primitive Culling

The mesh-shaders can do per-triangle culling. However, depending on the hardware and shading complexity this may speed things up or actually slow down. We use NV_mesh_shader rather than EXT_mesh_shader due to the ability to implement primitive culling more efficiently.

Shaded Results

4096 x 2048
RTX 6000 Ada



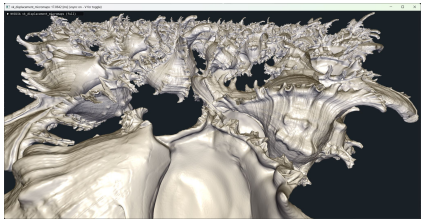
M Triangles	540	25	15
Render time [ms]	no LoD	LoD	LoD + occ culling
uncompressed ms	9.61	1.12	0.79
compressed ms mip decoder	12.75	1.21	0.85
compressed ms intrinsic decoder	18.41	1.56	1.07
compressed raytracing	1.35		

With LoD enabled the compressed data can be rendered at similar speeds to the uncompressed data. Without LoD the impact of using the hardware intrinsic decoding is substantial.

Raytracing is quite competitive for primary visibility due to “perfect” triangle occlusion culling.

Shaded Results

4096 x 2048
RTX 4060 Ti

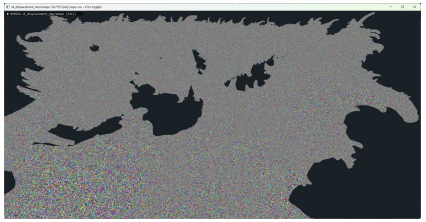


M Triangles	540	25	15
Render time [ms]	no LoD	LoD	LoD + occ culling
uncompressed ms	55.14	2.90	1.98
compressed ms mip decoder	57.30	3.25	2.24
compressed ms intrinsic decoder	72.52	4.54	3.00
compressed raytracing	4.73		

(same principle behavior as previous hardware configuration)

Visibility Results

4096 x 2048
RTX 6000 Ada

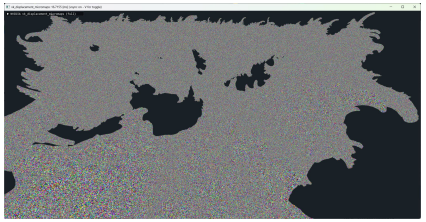


M Triangles	540	25	15
Render time [ms]	no LoD	LoD	LoD + occ culling
uncompressed ms	8.54	0.87	0.59
compressed ms mip decoder	10.14	0.92	0.63
compressed ms intrinsic decoder	13.02	0.97	0.66
compressed raytracing	1.29		

In the “id buffer” generation scenario the impact of the hardware intrinsic is lower and especially with LoD both decoder types are closer. This allows using the opaque data from the raytracing representation directly.

Visibility Results

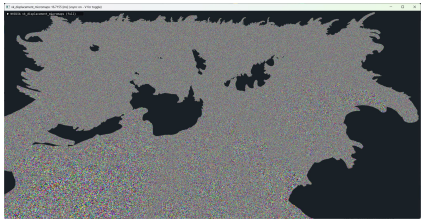
4096 x 2048
RTX 4060 Ti



M Triangles	540	25	15
Render time [ms]	no LoD	LoD	LoD + occ culling
uncompressed ms	55.33	2.62	1.77
compressed ms mip decoder	56.07	2.72	1.87
compressed ms intrinsic decoder	56.41	3.00	2.09
compressed raytracing	4.55		

(same principle behavior as previous hardware configuration)

Compute vs Mesh Shader

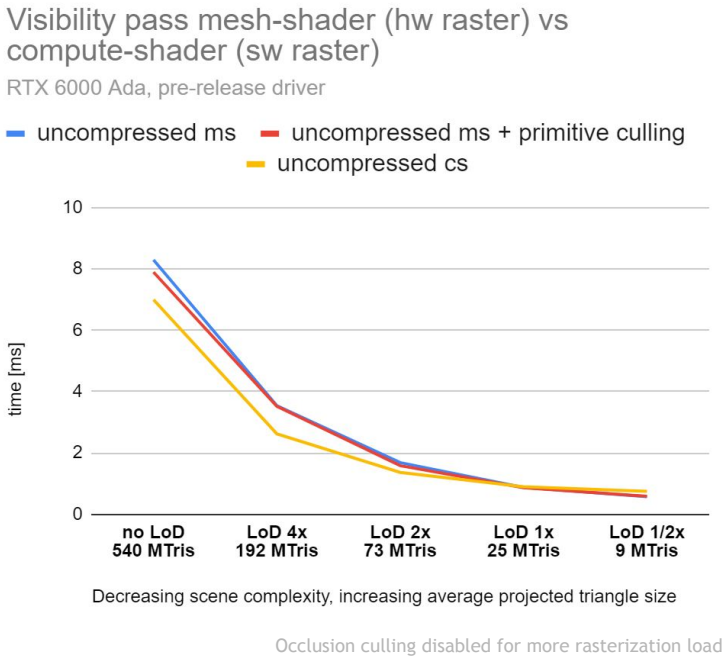


4096 x 2048
RTX 6000 Ada

Whether software rasterization (compute) or hardware rasterization (mesh shader) is faster, depends on the number of sub-pixel triangles.

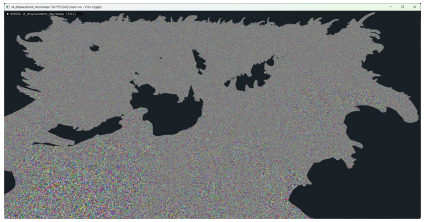
LoD 1x targets around 1 pixel per triangle and on NVIDIA hardware is still pretty fast to be rastered via hardware units using mesh shaders. LoD 2x means we push the LoD decision further out, so more sub-pixel triangles, while 1/2 x means earlier, larger triangles (this risks visible cracks).

Compressed data was a tad slower than uncompressed but had similar behavior.



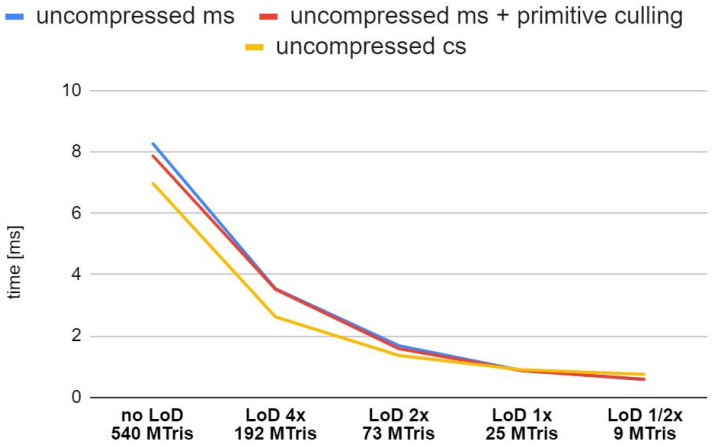
Compute vs Mesh Shader

4096 x 2048



Visibility pass mesh-shader (hw raster) vs compute-shader (sw raster)

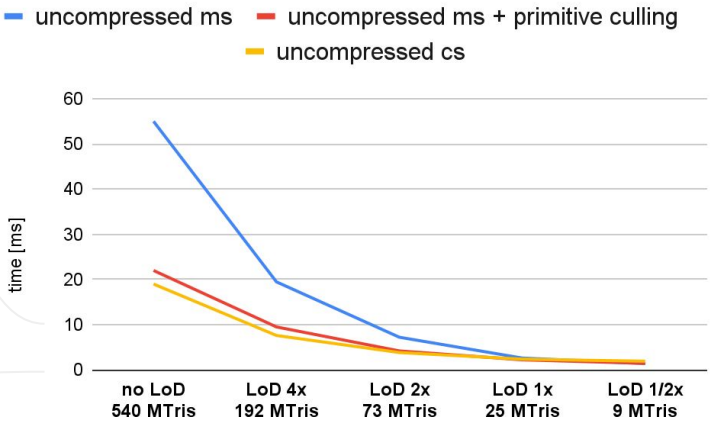
RTX 6000 Ada, pre-release driver



Decreasing scene complexity, increasing average projected triangle size

Visibility pass mesh-shader (hw raster) vs compute-shader (sw raster)

RTX 4060 Ti, pre-release driver



Decreasing scene complexity, increasing average projected triangle size

Occlusion culling disabled for more rasterization load

Performance Conclusions

Data Representation

- Uncompressed is faster, but “base w. mip decoder” can be close when using dynamic lod.
- 64-block format is 45 x uncompressed unorm11, easy to fetch (no special decoding/encoding)
- Hardware intrinsics are an option if visibility pass in combination with dynamic LoD is used or if memory constrained.

Compute vs Mesh-Shading

- Depending on dynamic LoD usage can yield slightly larger triangles which favors mesh-shaders
- May want to use 2x 64-bit atomics, as reconstruction of base-triangle barycentric UVs would be ugly.
- Dual pass compute needs a lot transient data between task and mesh pass. “Split” variant needs less.
  (12 bytes per visible rasterization warp = 64 micro triangles max, {u32 instanceID, baseID, binPackInfo})

Dynamic LoD

- Higher subdivision levels allow more LoD and enable more vertex re-use, better utilization
- If subdivision is very low (1 or 4 micro-triangles per base-triangle), it’s better to render mesh by other means
- Bin packing is great for perf, half performance when disabling it and handling base-triangles one warp at a time.

Backup

Note: Naming Issue

There was a bit of renaming in the documentation and code, so the use of the words “bary” vs “umesh” are described here, in case you stumble upon them.

bary == uncompressed: rendering the uncompressed values of a base-triangle as they are generated from baking. Serves as reference to benchmark decoding performance.

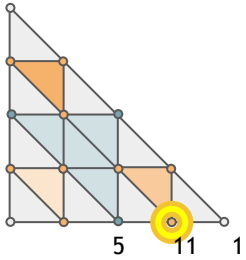
Example: for 256 micro triangles = 153 values (typically 16 bit unorm displacements) = 2448 bit

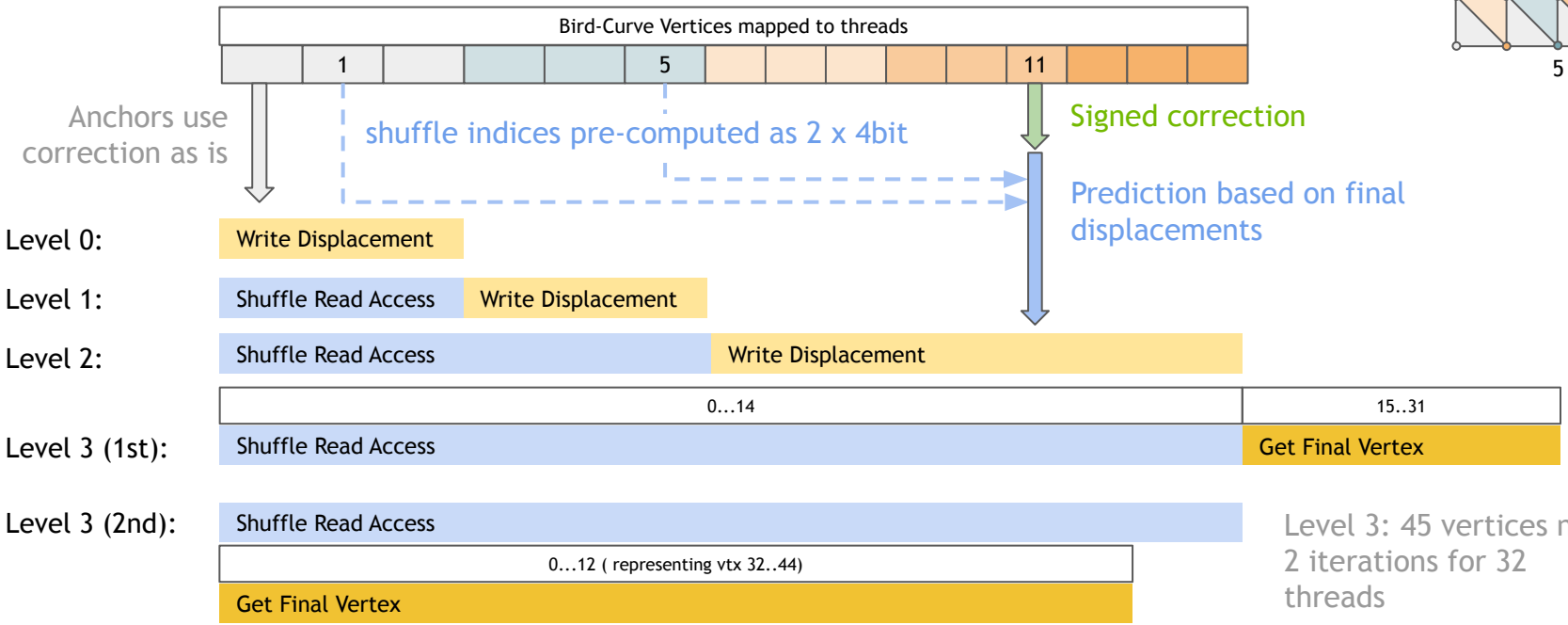
umesh == compressed: rendering the block-compressed representation where a base-triangle is split into sub-triangles . Each sub-triangle is represented by a single compressed block of 1024 or 512 bit.

Example: for 256 micro triangles = 4 x 512 bit (each 64 microtris) **or** 1 x 1024 bit.

Manual Decoder Implementation

- Iterate subdiv levels, shift active threads, read previous, write new





Level 3: 45 vertices need 2 iterations for 32 threads