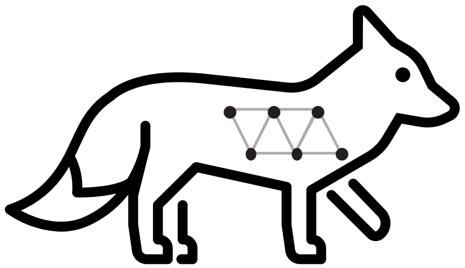
MESH SHADERS IN AMD RDNA™ 3 ARCHITECTURE

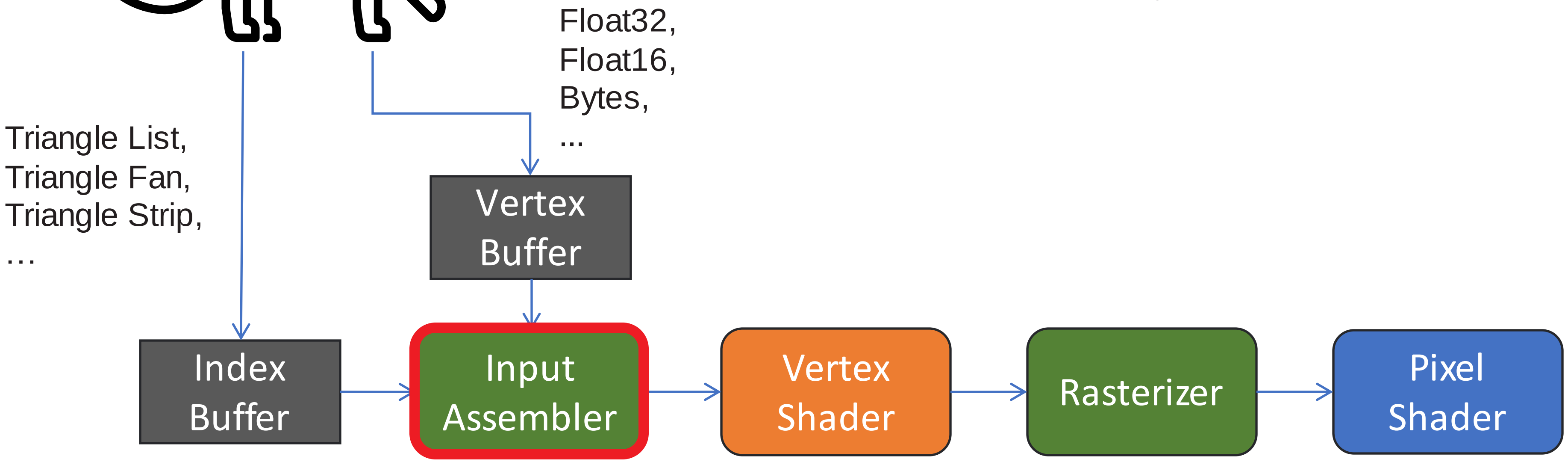
MAX OBERBERGER, LOU KRAMER

AMD
together we advance_

TRADITIONAL GEOMETRY PIPELINE

- The traditional geometry pipeline was used for over two decades
- Limited flexibility due to fixed function blocks

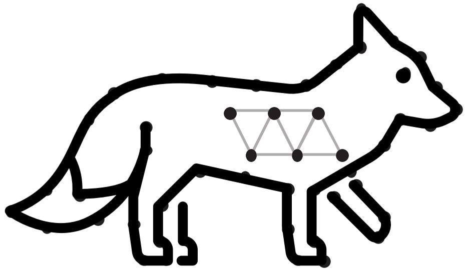


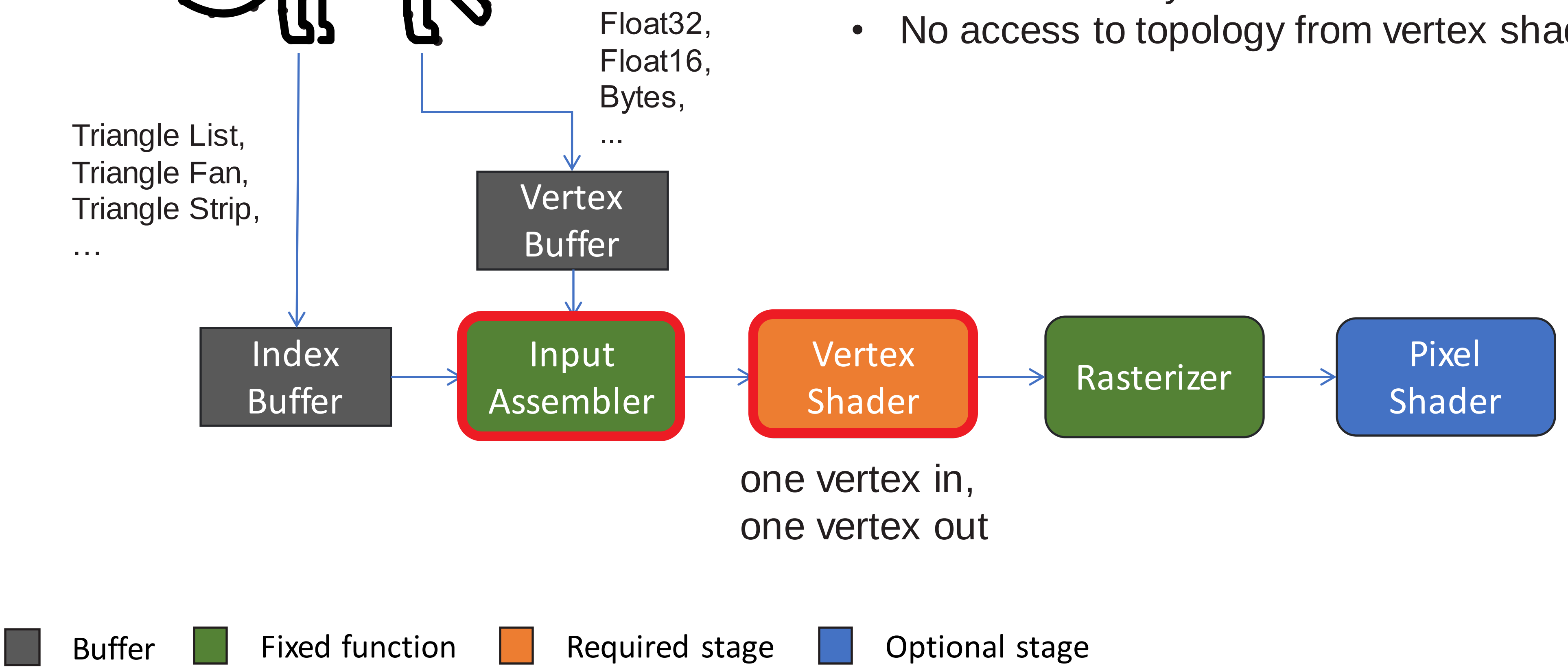


Buffer | Fixed function | Required stage | Optional stage

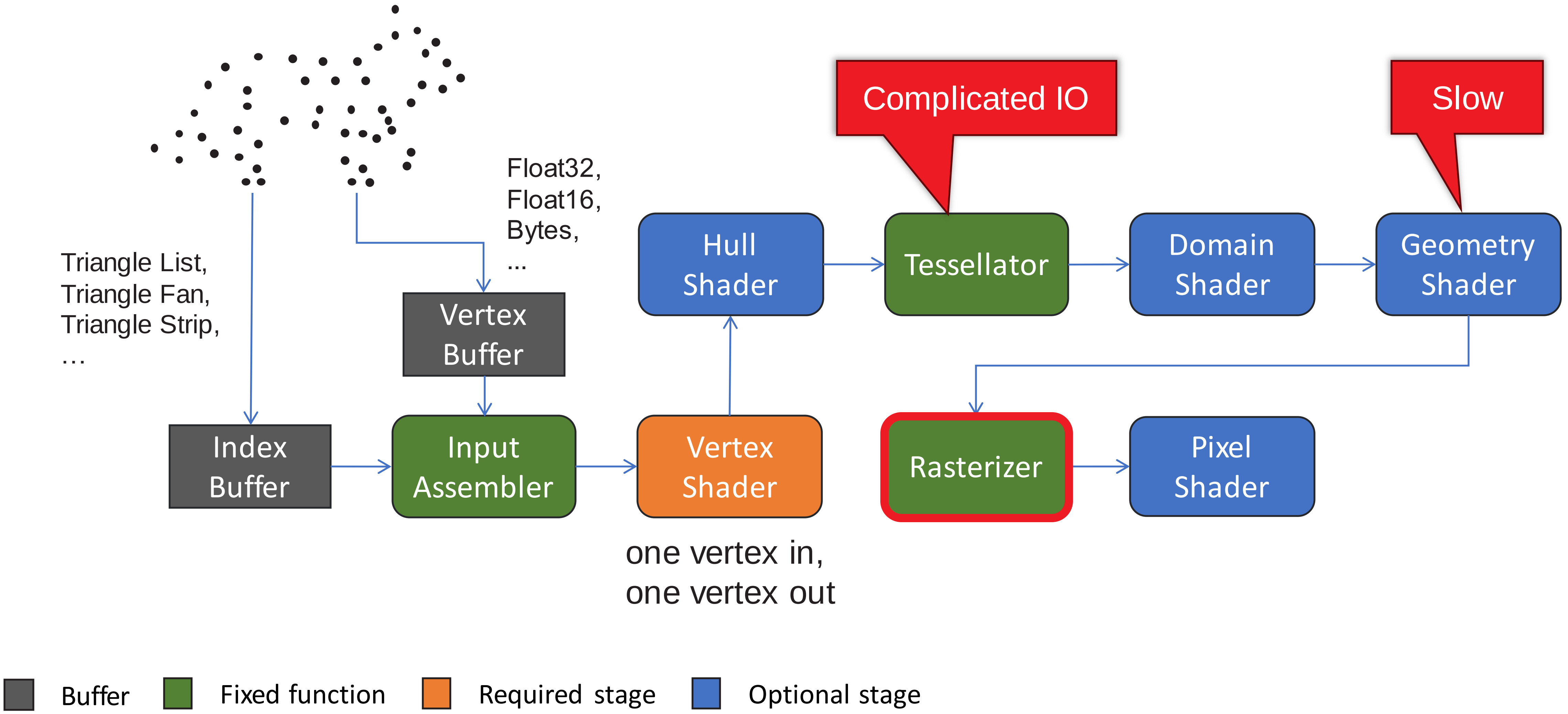
TRADITIONAL GEOMETRY PIPELINE

- The traditional geometry pipeline was used for over two decades
- Limited flexibility due to fixed function blocks
- No access to topology from vertex shader

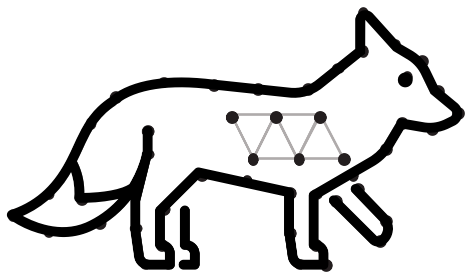




TRADITIONAL GEOMETRY PIPELINE

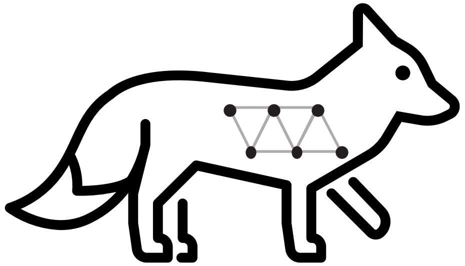


MESH SHADER PIPELINE



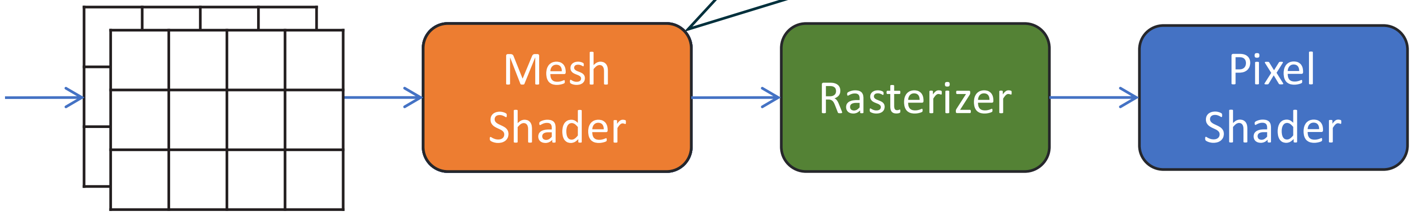
Buffer Fixed function Required stage Optional stage

MESH SHADER PIPELINE



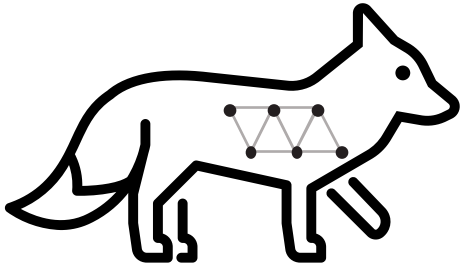
```
[NumThreads(128, 1, 1)]
[OutputTopology("triangle")]
void MeshShader(
    out indices uint3 tris[MAX_TRIANGLES],
    out vertices VertexOutput verts[MAX_VERTICES],
```

```
DispatchMesh(4, 3, 2)
```



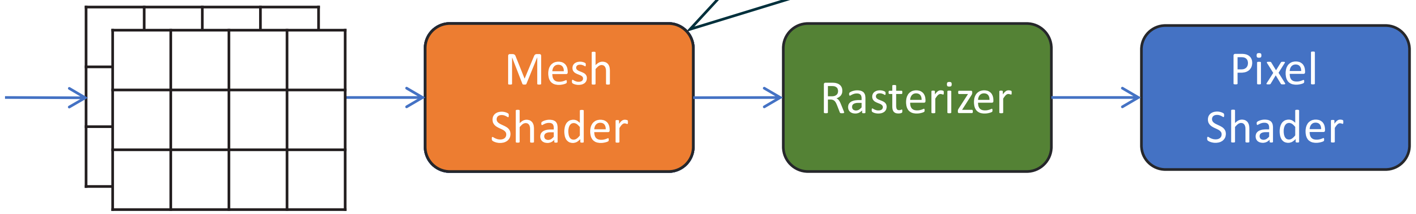
Buffer   Fixed function   Required stage   Optional stage

MESH SHADER PIPELINE



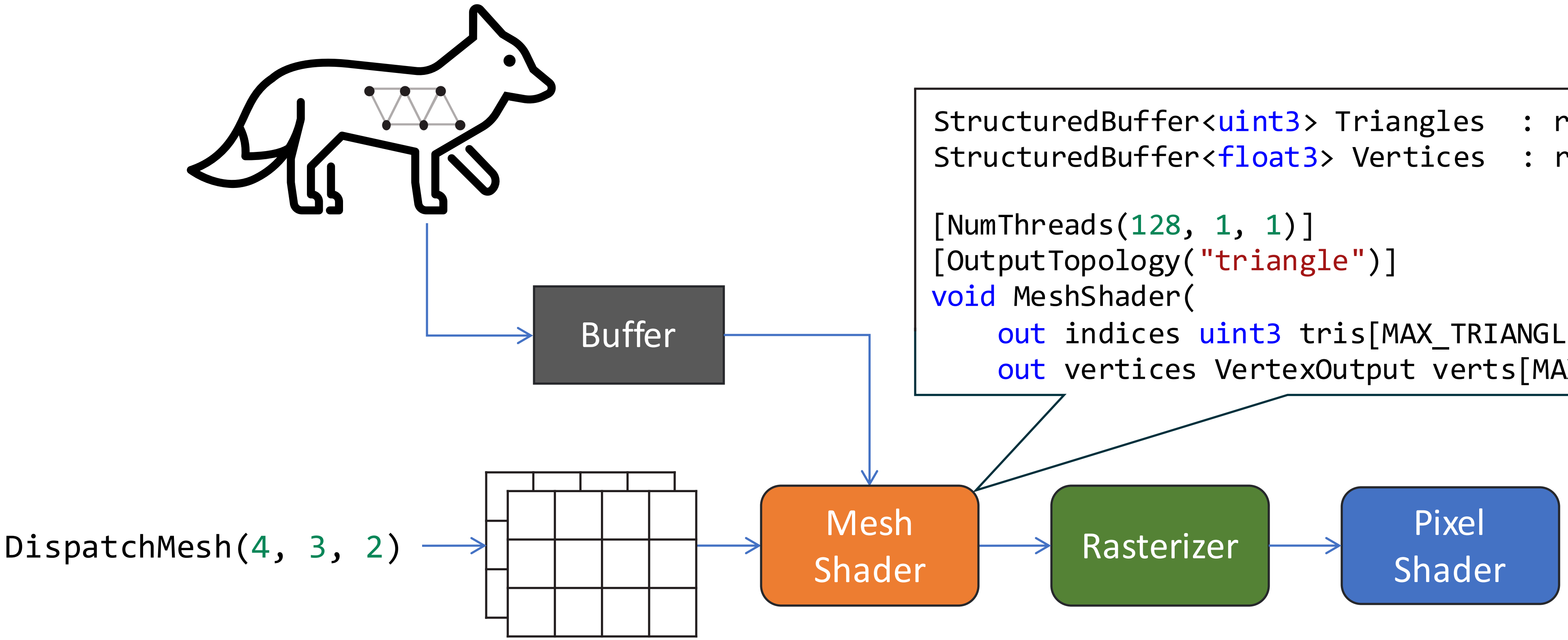
```
[NumThreads(128, 1, 1)]
[OutputTopology("triangle")]
void MeshShader(
    out indices uint3 tris[MAX_TRIANGLES],
    out vertices VertexOutput verts[MAX_VERTICES],
    uint threadId : SV_GroupThreadID) {

    ...

    verts[threadId] = LoadVertex(threadId);
    tris[threadId] = LoadTriangle(threadId);
```

`DispatchMesh(4, 3, 2)`



Buffer | Fixed function | Required stage | Optional stage

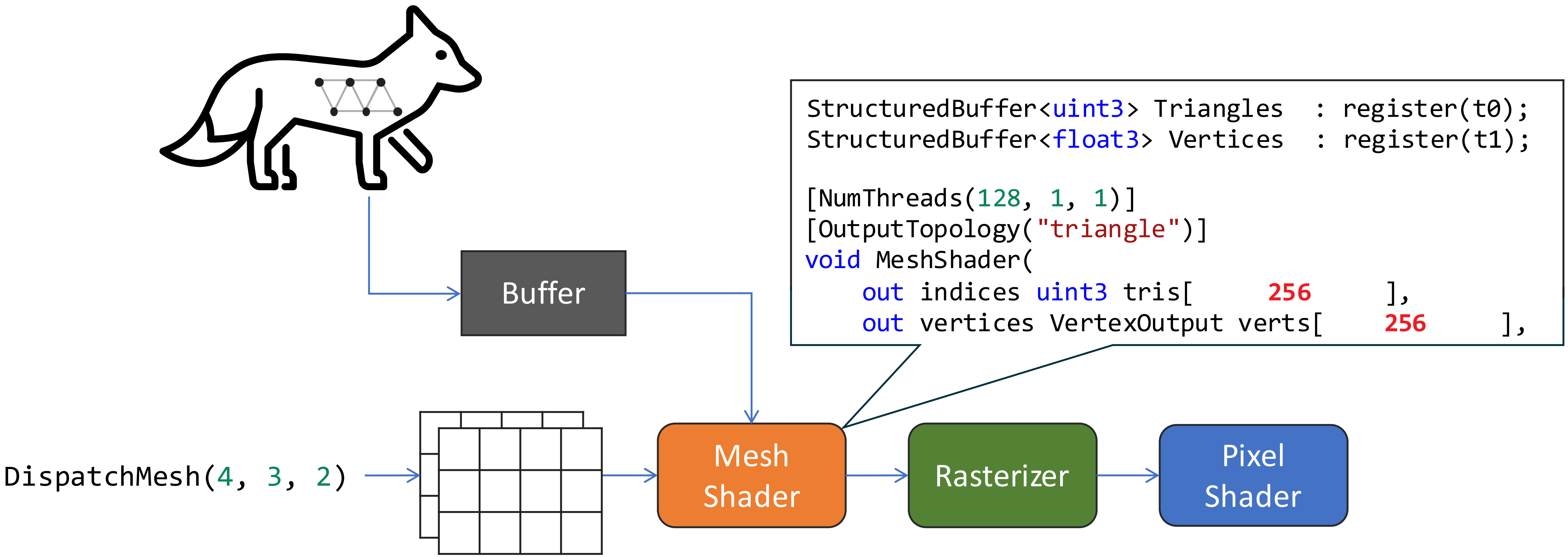
MESH SHADER PIPELINE




```
StructuredBuffer<uint3> Triangles  : register(t0);
StructuredBuffer<float3> Vertices  : register(t1);

[NumThreads(128, 1, 1)]
[OutputTopology("triangle")]
void MeshShader(
    out indices uint3 tris[MAX_TRIANGLES],
    out vertices VertexOutput verts[MAX_VERTICES],
```

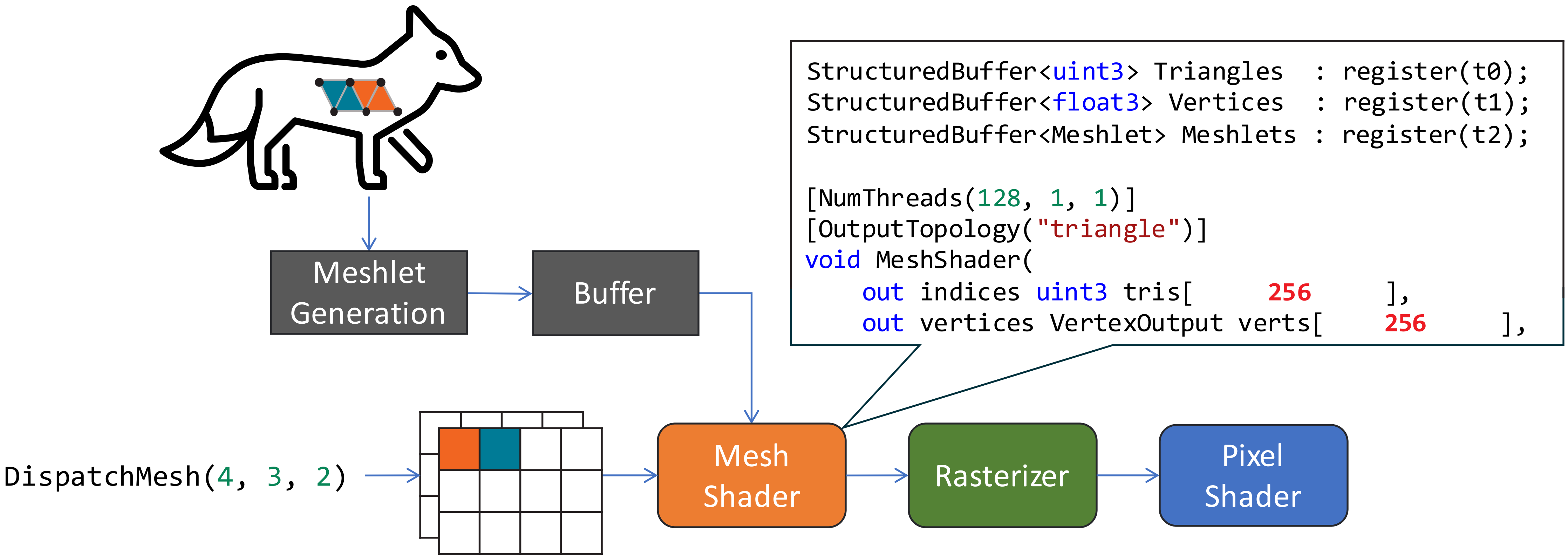
Buffer   Fixed function   Required stage   Optional stage

MESH SHADER PIPELINE



Buffer | Fixed function | Required stage | Optional stage

MESH SHADER PIPELINE




```
StructuredBuffer<uint3> Triangles  : register(t0);
StructuredBuffer<float3> Vertices  : register(t1);
StructuredBuffer<Meshlet> Meshlets : register(t2);

[NumThreads(128, 1, 1)]
[OutputTopology("triangle")]
void MeshShader(
    out indices uint3 tris[     256     ],
    out vertices VertexOutput verts[    256     ],
```

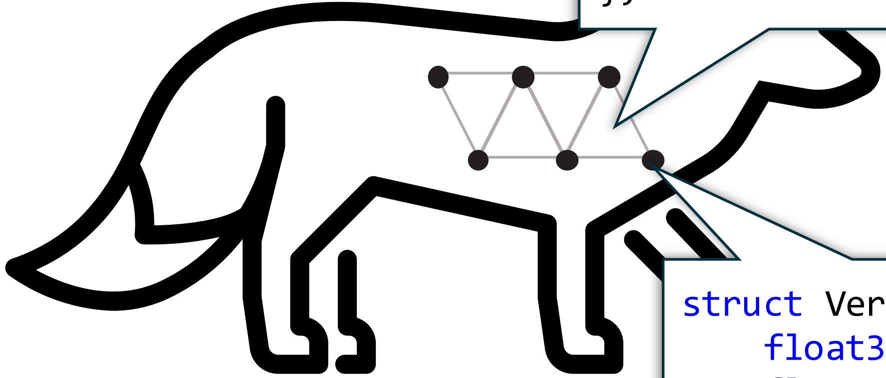
Buffer | Fixed function | Required stage | Optional stage

SO YOU WANT TO WRITE A MESH SHADER

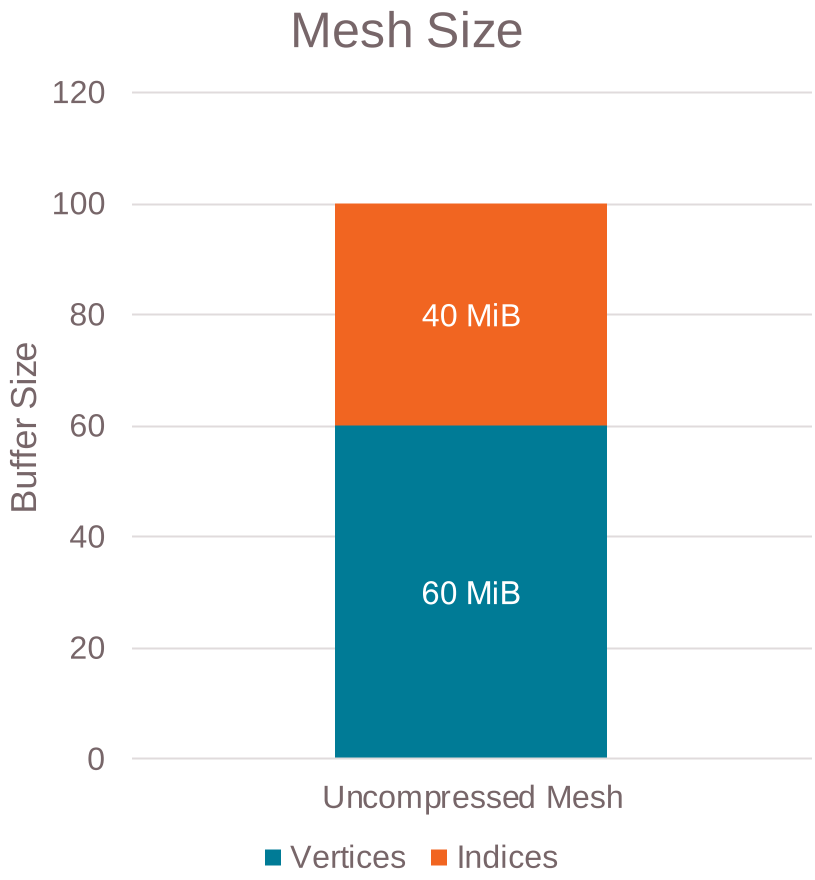
```
[NumThreads(128, 1, 1)]
[OutputTopology("triangle")]
void MeshShader(
    uint threadId : SV_GroupThreadID,
    out indices uint3 tris[MAX_TRIANGLES],
    out vertices VertexOutput verts[MAX_VERTICES]) {

    // Now what?

}
```

COMPRESSION

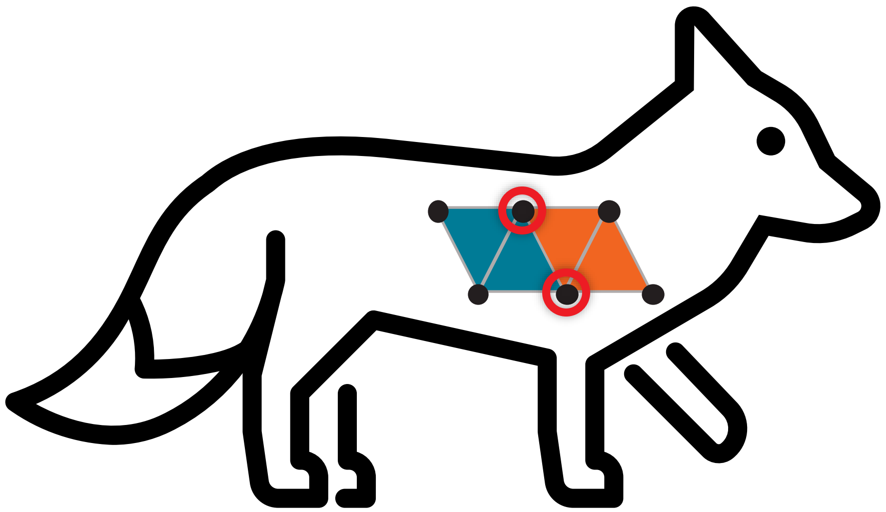
```
struct Triangle {
    uint32_t v0;
    uint32_t v1;
    uint32_t v2;
};
```

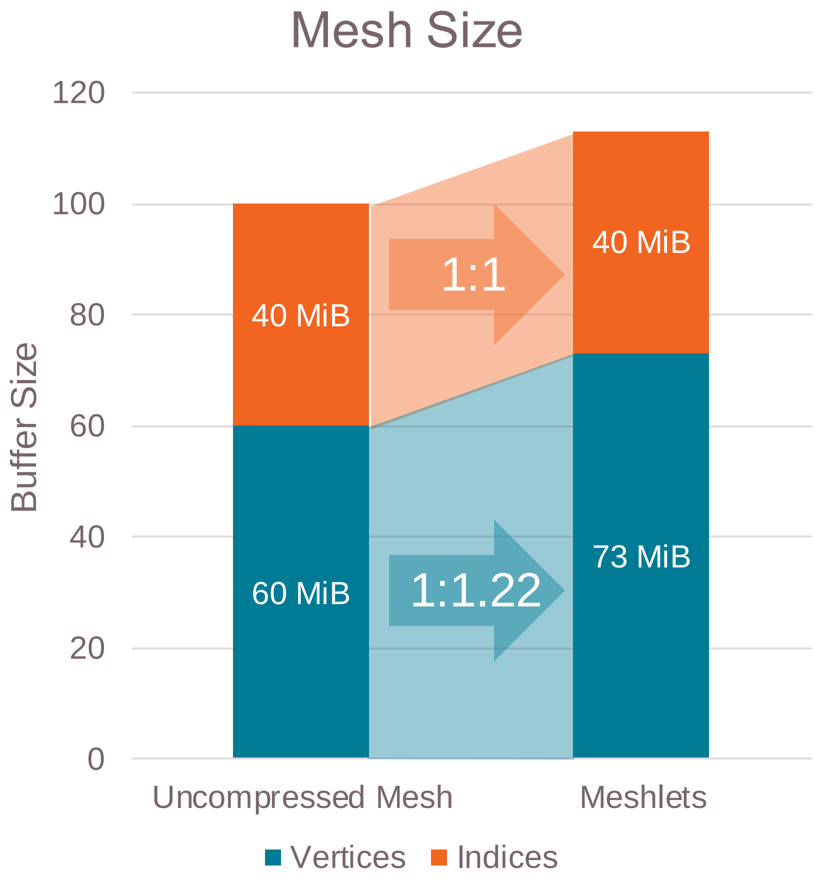


```
struct Vertex {
    float3 position;
    float3 normal;
    float2 texCoord;
};
```

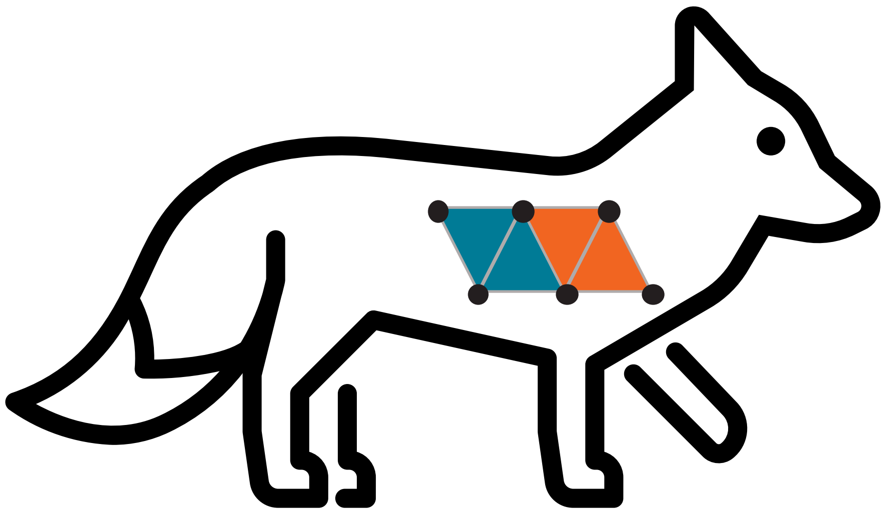


COMPRESSION





COMPRESSION

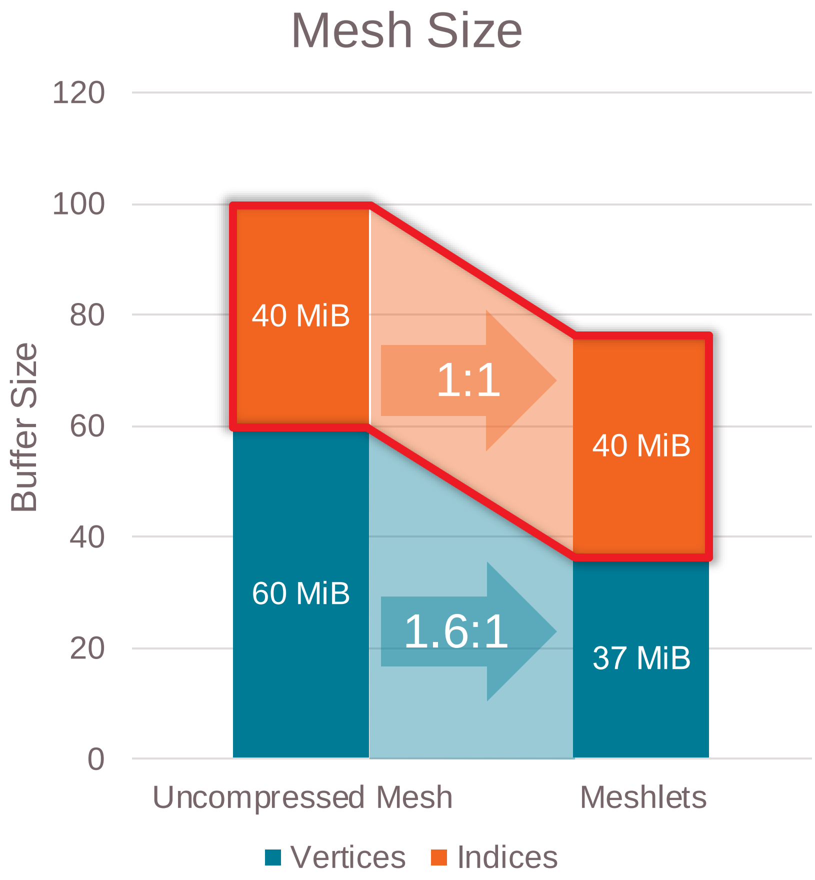


18 bit

Global Quantization

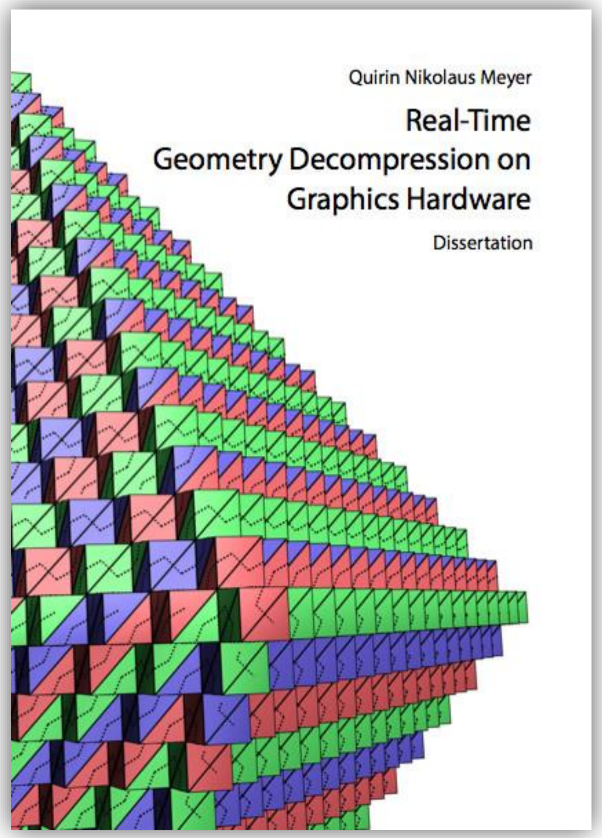
~16 bit

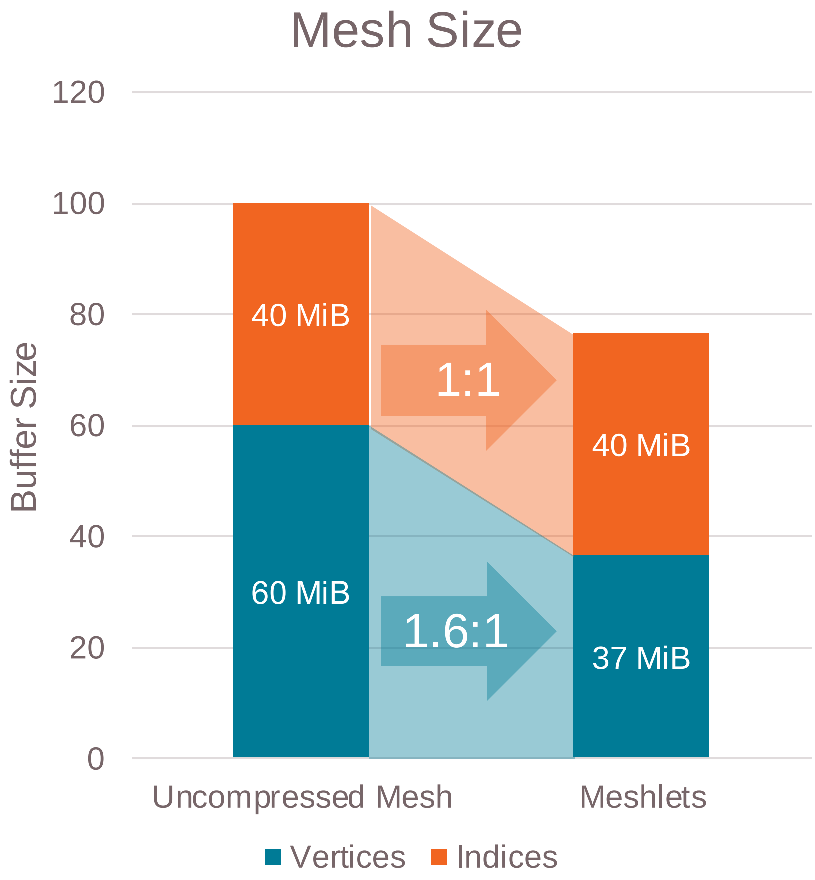
Local Quantization



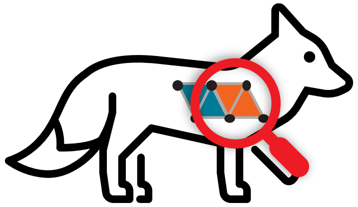
COMPRESSION

2012



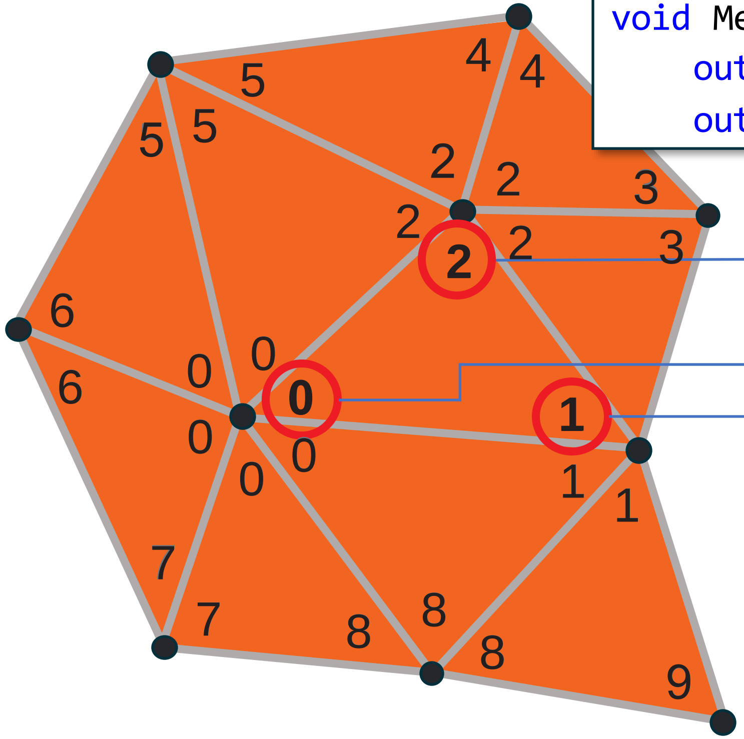


COMPRESSION



Vertex & Triangle counts are limited to 256 elements each

```
[NumThreads(128, 1, 1)]
[OutputTopology("triangle")]
void MeshShader(
    out indices uint3 tris[      256      ],
    out vertices VertexOutput verts[     256      ],
```

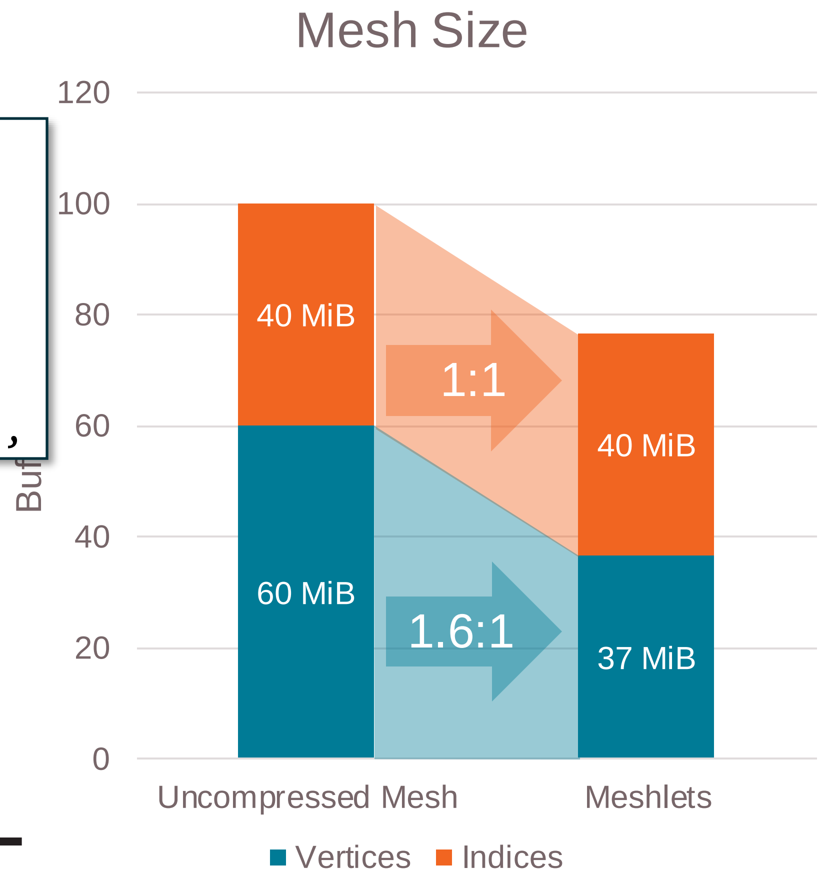


32 Bit Index

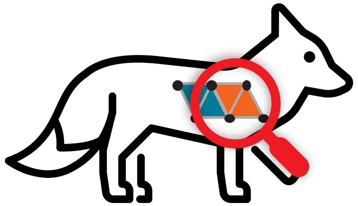
32 Bit Index

32 Bit Index

96 Bits per Triangle

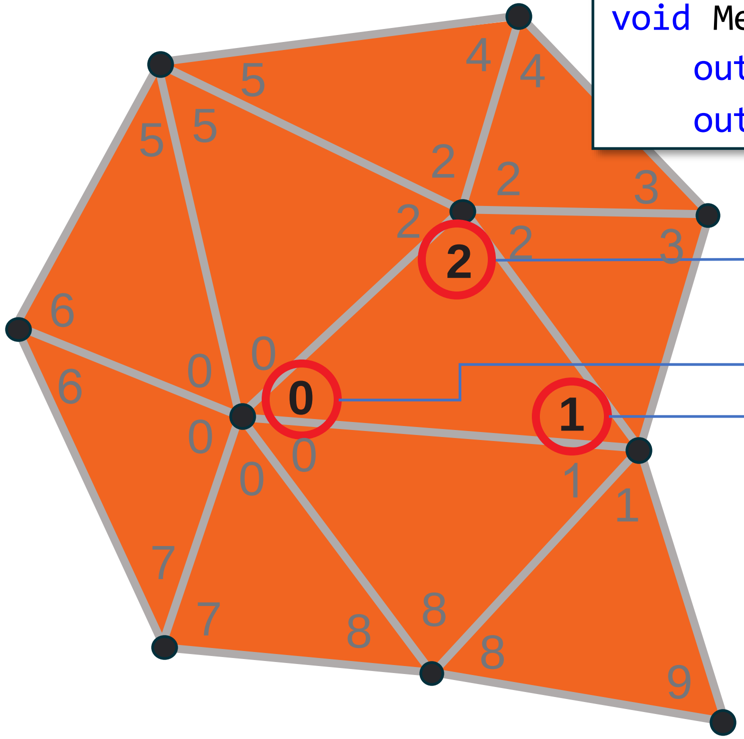


COMPRESSION



Vertex & Triangle counts are limited to 256 elements each

```
[NumThreads(128, 1, 1)]
[OutputTopology("triangle")]
void MeshShader(
    out indices uint3 tris[      256     ],
    out vertices VertexOutput verts[    256     ],
```

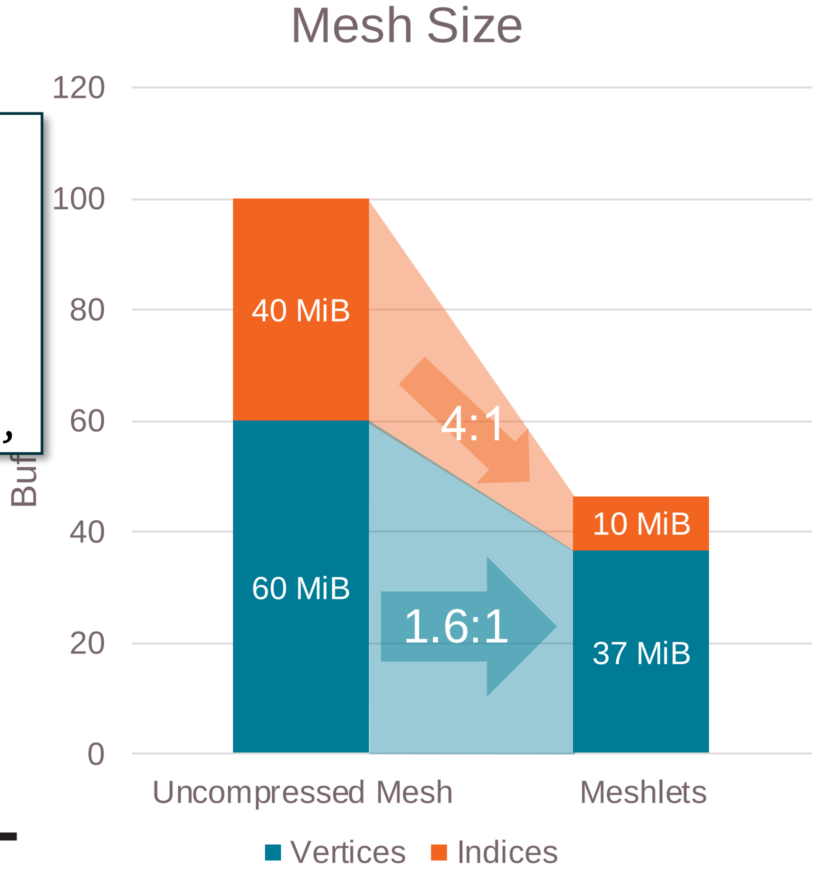


8 Bit Index

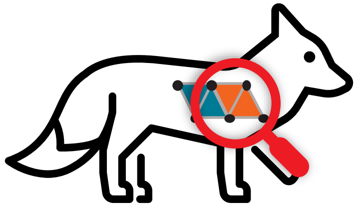
8 Bit Index

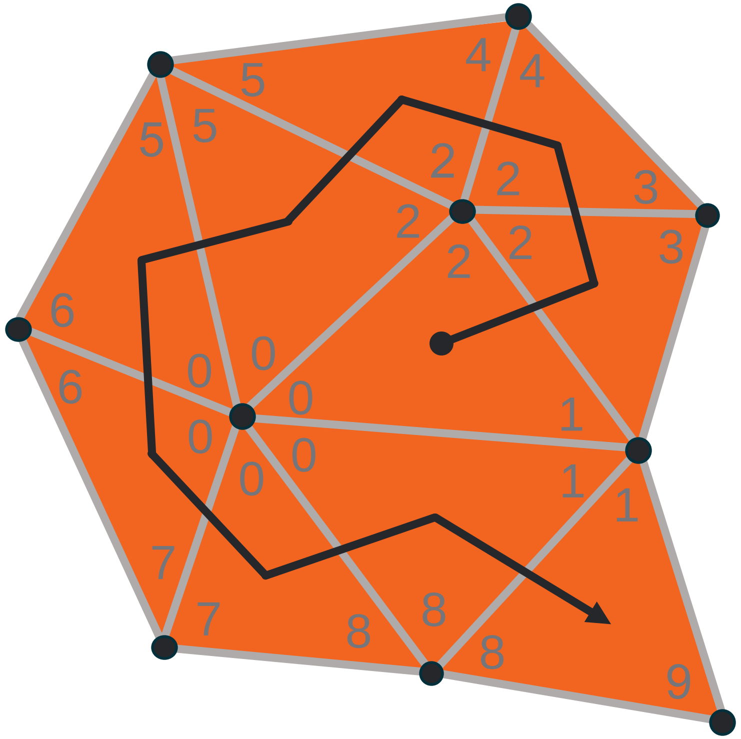
8 Bit Index

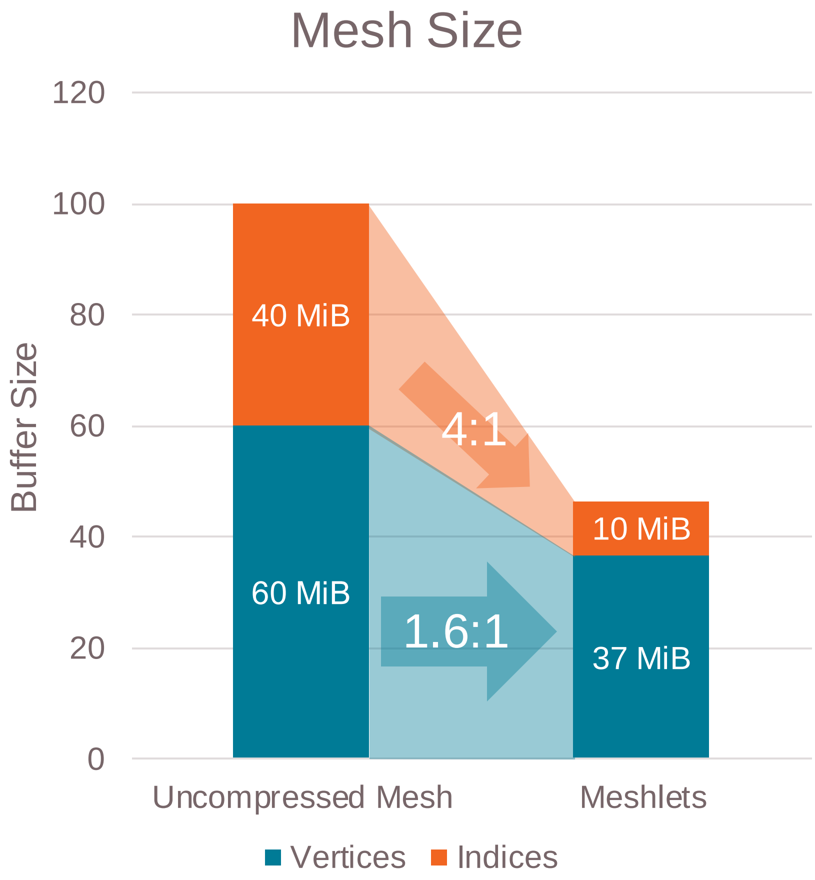
24 Bits per Triangle



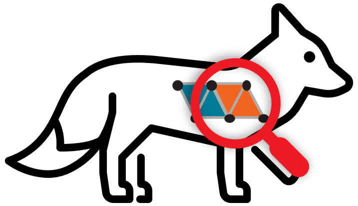
COMPRESSION

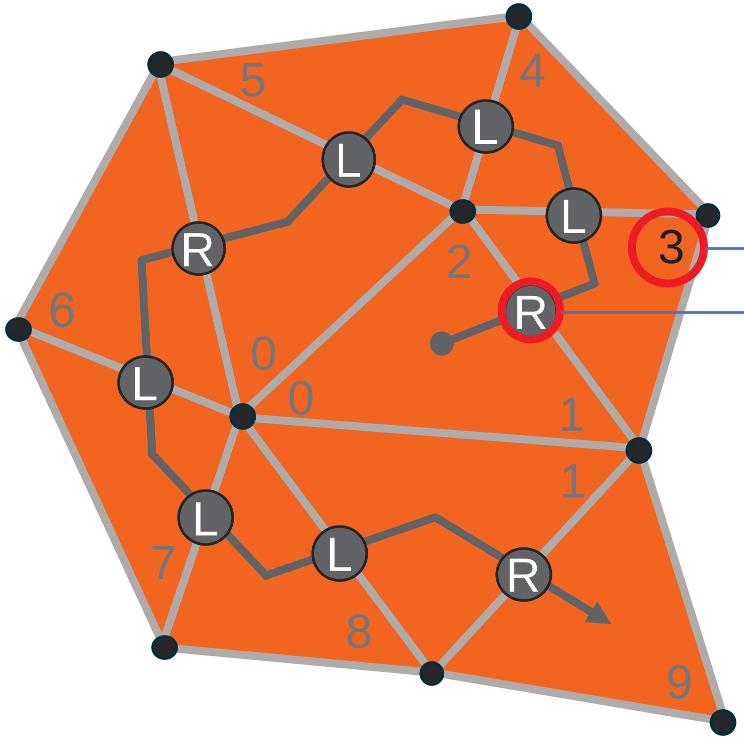






COMPRESSION

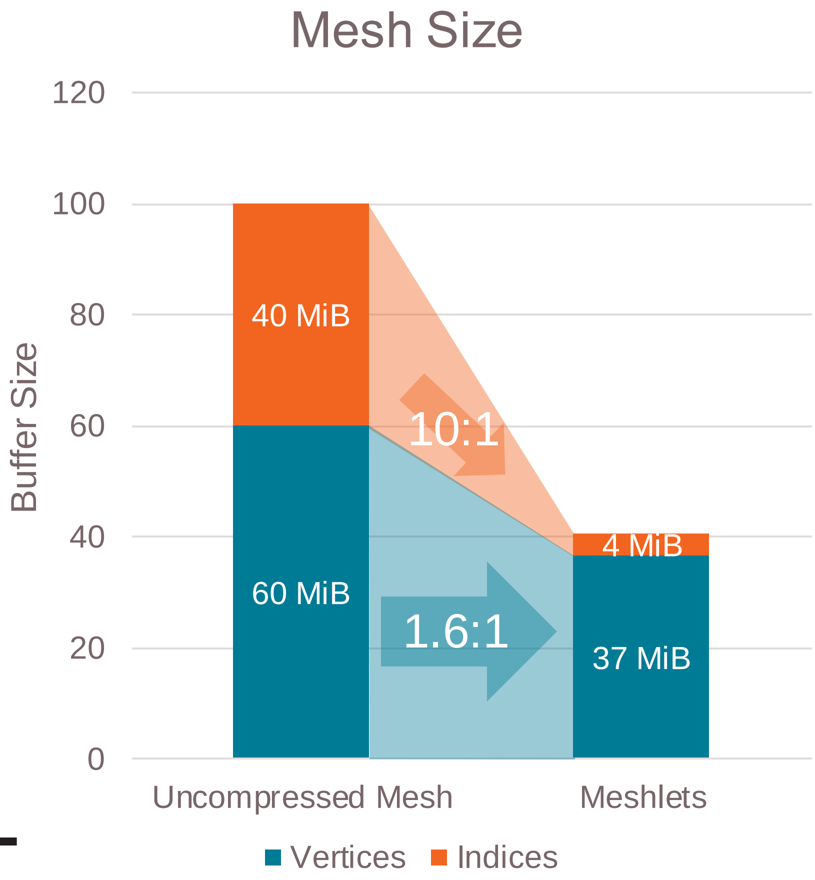




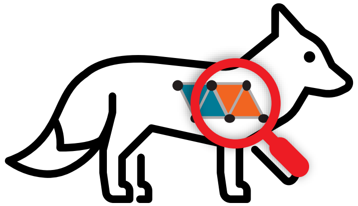
8 Bit Index

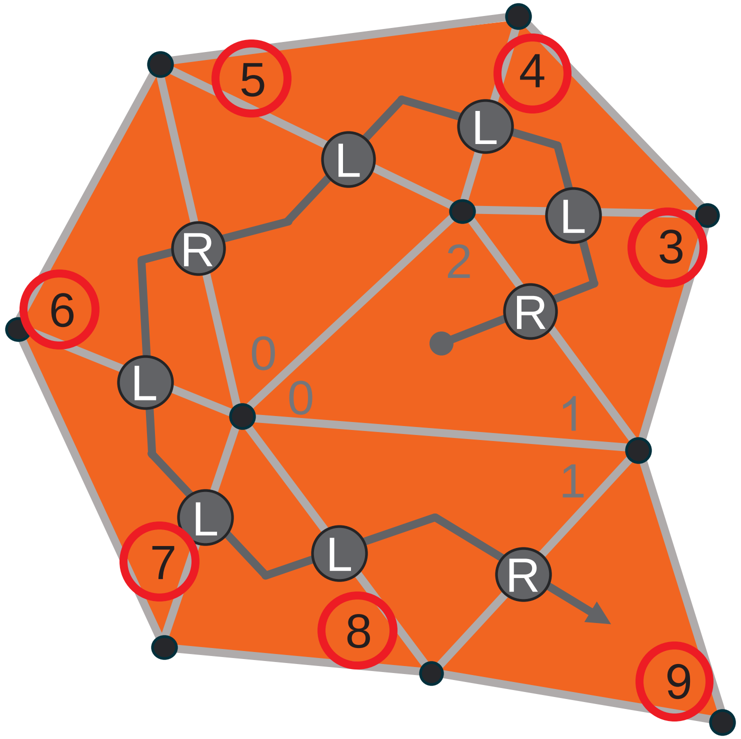
1 Bit Code

9 Bits per Triangle



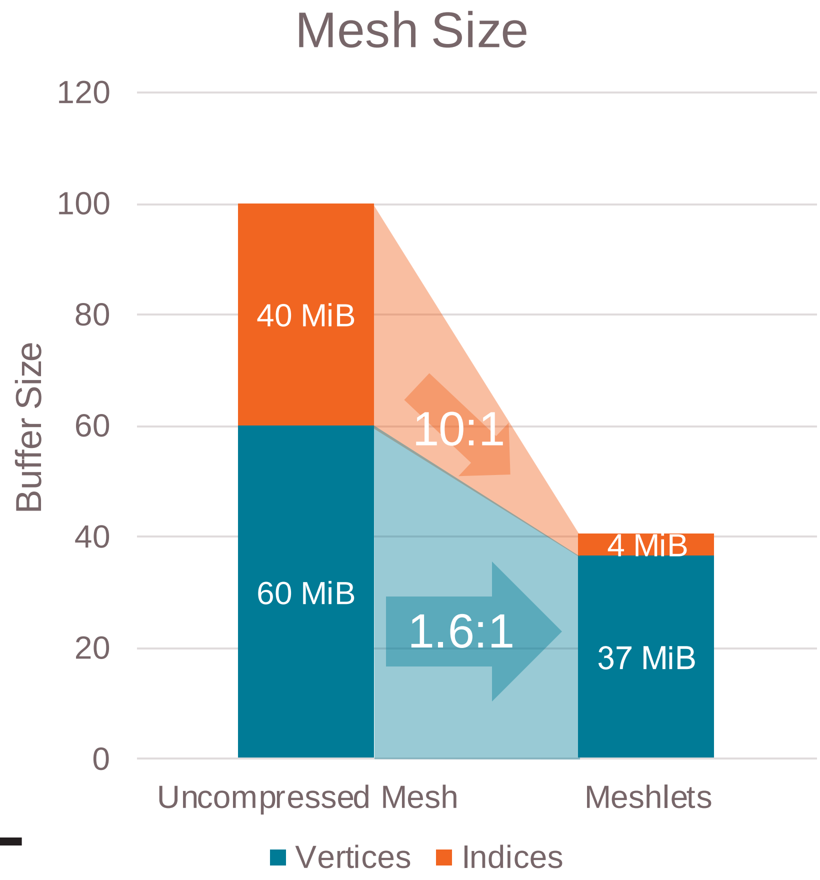
COMPRESSION



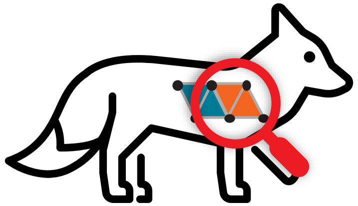


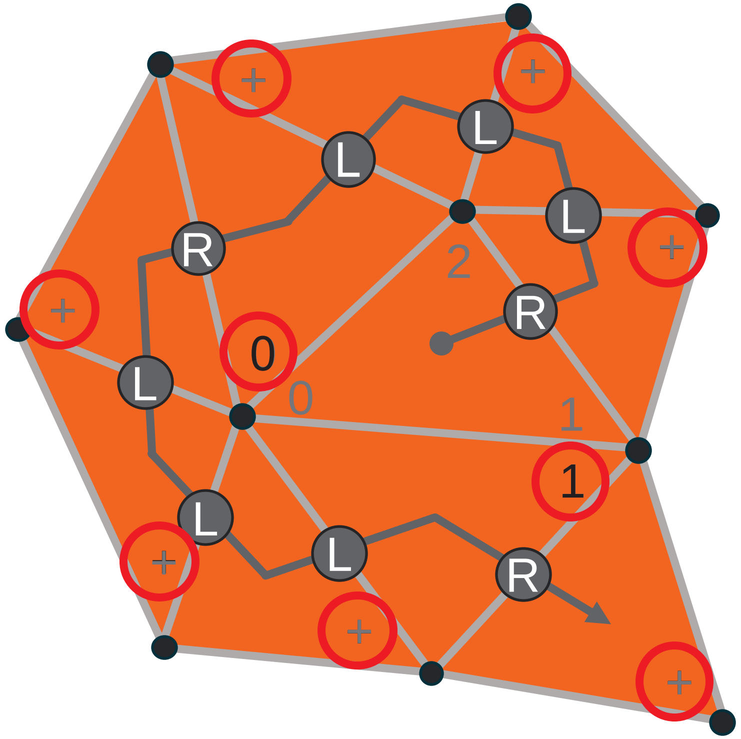
8 Bit Index
1 Bit Code

9 Bits per Triangle



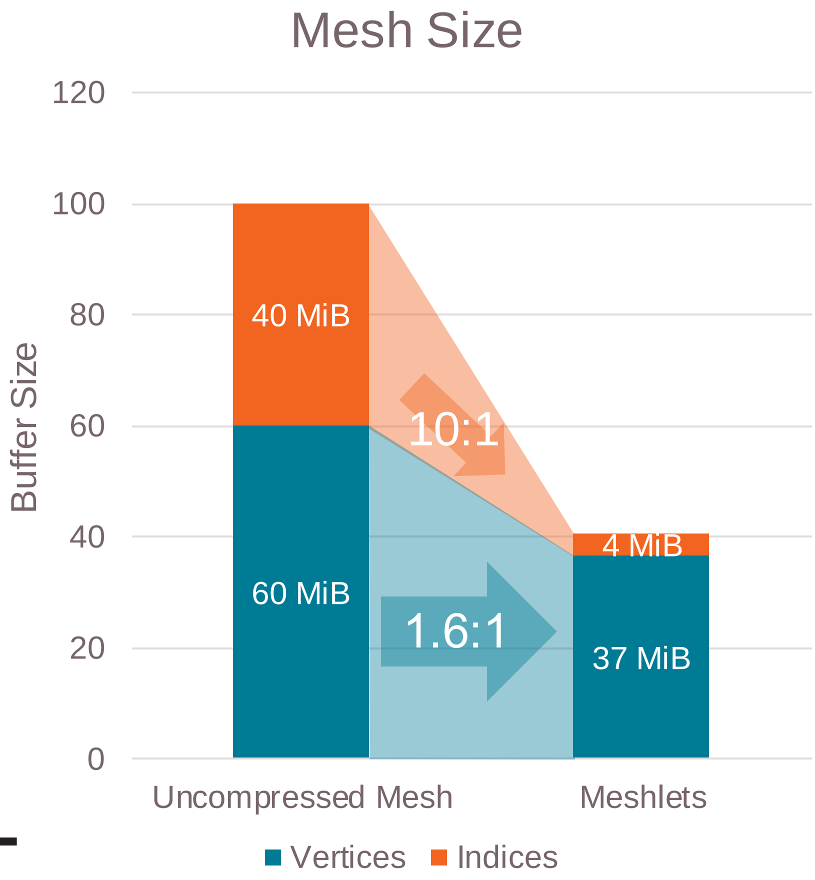
COMPRESSION



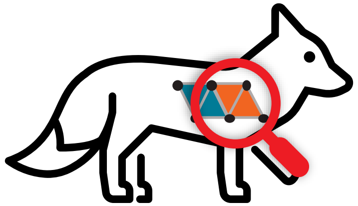


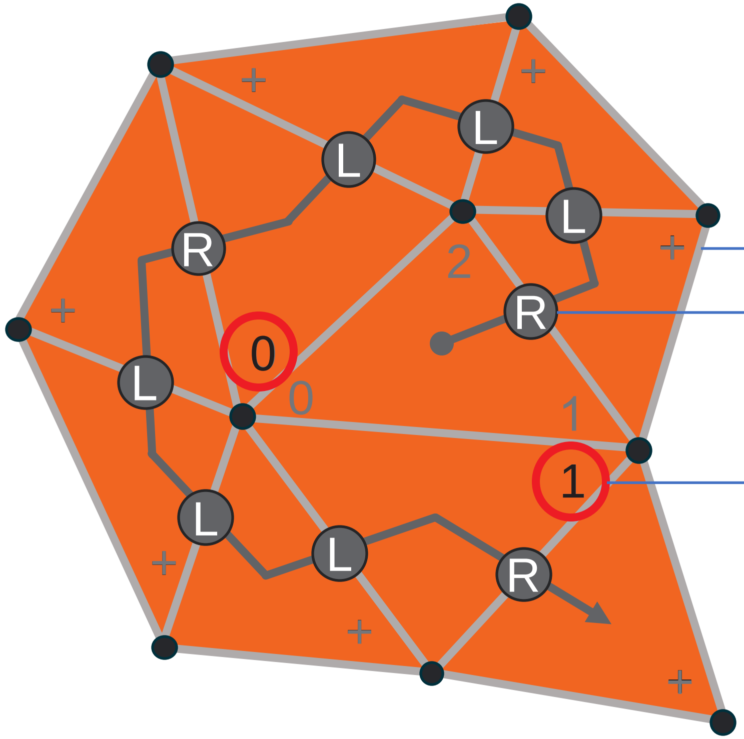
8 Bit Index
1 Bit Code

9 Bits per Triangle



COMPRESSION



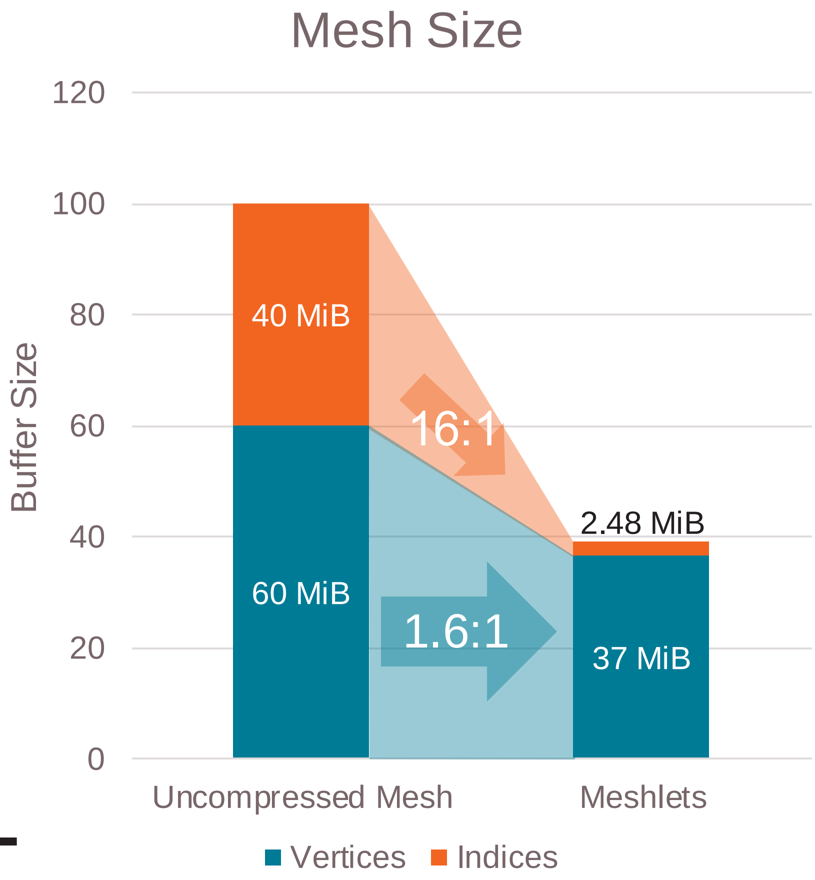


1 Bit Code

1 Bit Code

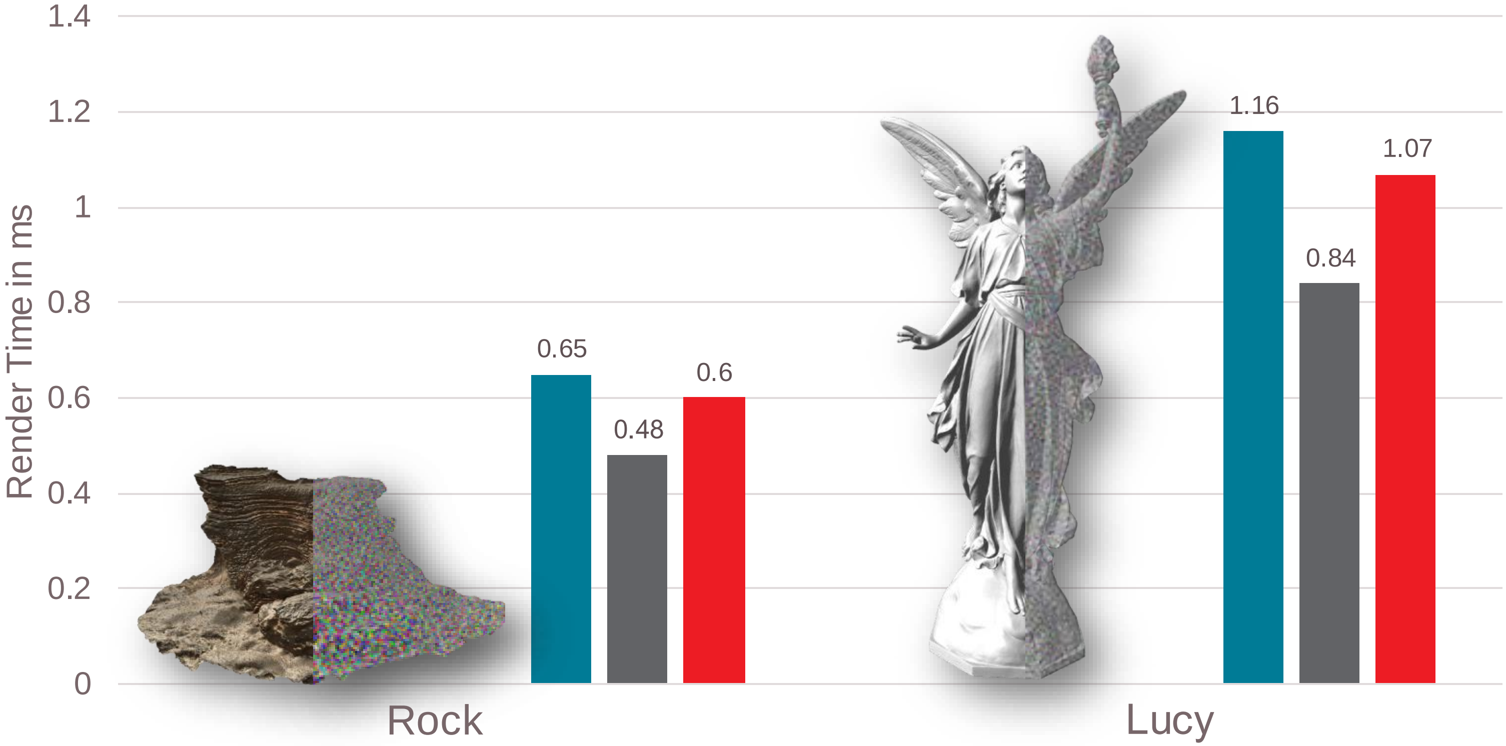
~3 Bit Index

~5 Bits per Triangle



COMPRESSION

Render Time

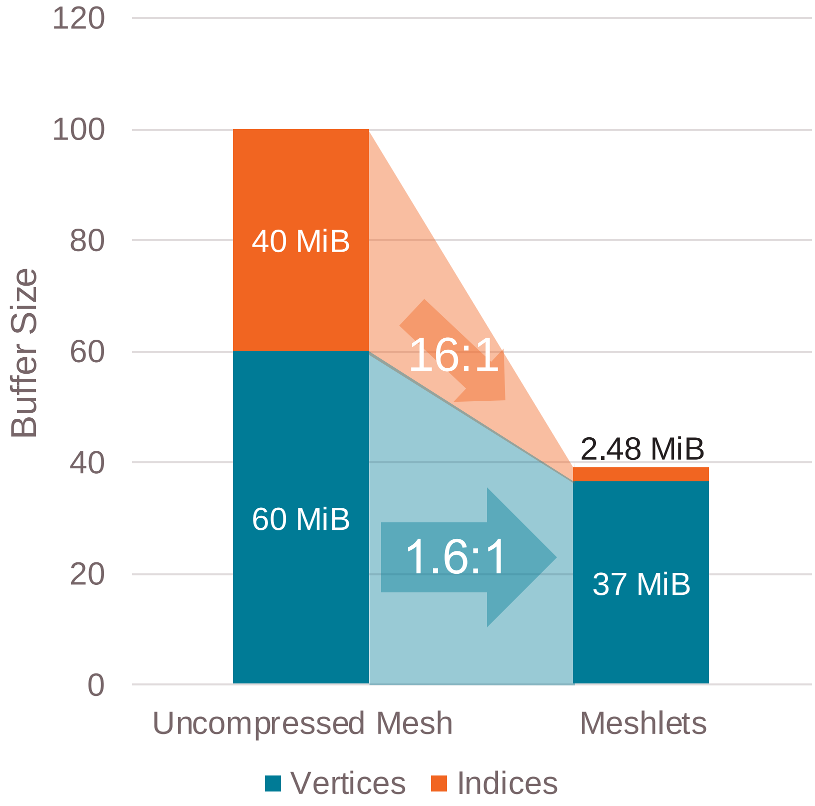


15.5 M Triangles
7.8 M Vertices
75 k Meshlets

28 M Triangles
14 M Vertices
136 k Meshlets

Vertex Pipeline Meshlets Compressed Meshlets

Mesh Size



GPU: AMD Radeon™ RX 7900 XTX; driver version: 24.1.1; Meshes: Layered Rock by Aixterior, Lucy by Stanford Computer Graphics Laboratory

COMPRESSION

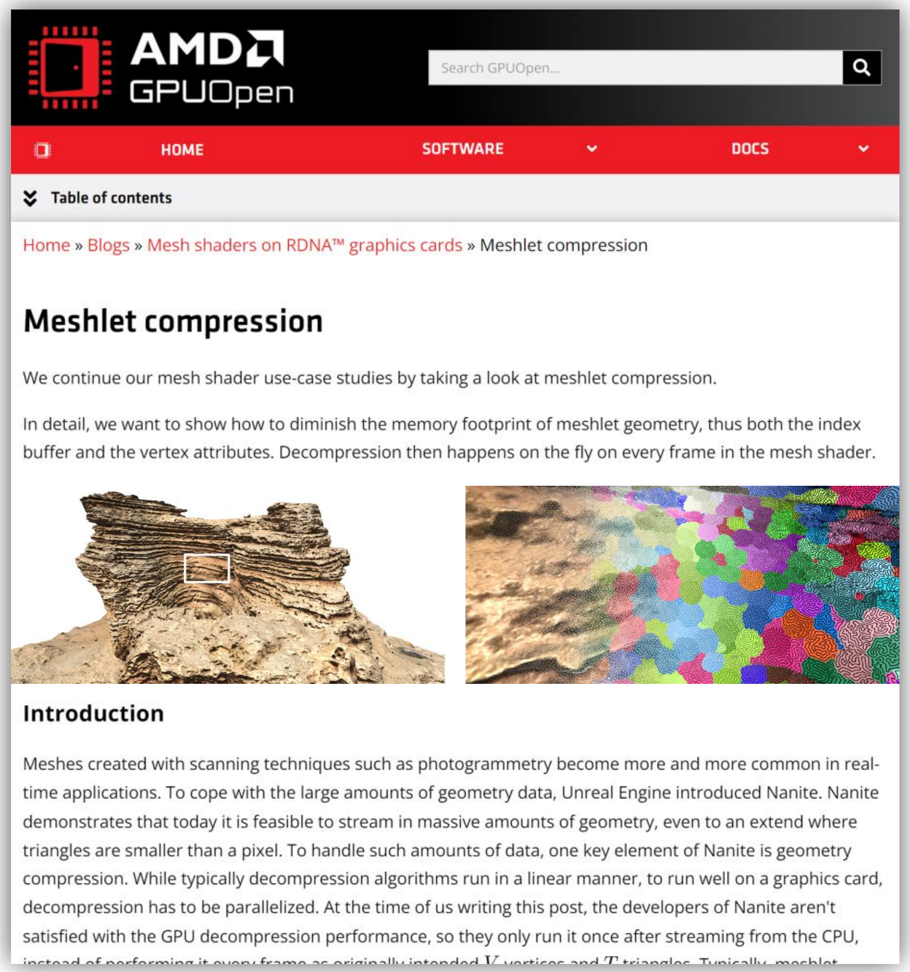
Keep an eye on GPUOpen.com

Follow us on X
https://twitter.com/GPUOpen

Follow us on Mastodon
https://mastodon.gamedev.place/@gpuopen

知乎

Follow us on Zhihu
https://www.zhihu.com/org/gpuopen-7



Meshlet compression

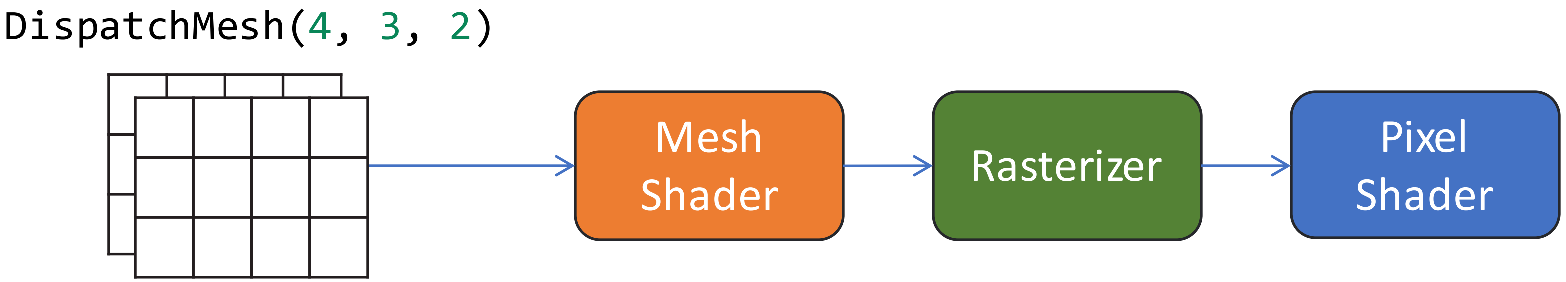
We continue our mesh shader use-case studies by taking a look at meshlet compression.

In detail, we want to show how to diminish the memory footprint of meshlet geometry, thus both the index buffer and the vertex attributes. Decompression then happens on the fly on every frame in the mesh shader.

Introduction

Meshes created with scanning techniques such as photogrammetry become more and more common in real-time applications. To cope with the large amounts of geometry data, Unreal Engine introduced Nanite. Nanite demonstrates that today it is feasible to stream in massive amounts of geometry, even to an extend where triangles are smaller than a pixel. To handle such amounts of data, one key element of Nanite is geometry compression. While typically decompression algorithms run in a linear manner, to run well on a graphics card, decompression has to be parallelized. At the time of us writing this post, the developers of Nanite aren't satisfied with the GPU decompression performance, so they only run it once after streaming from the CPU,

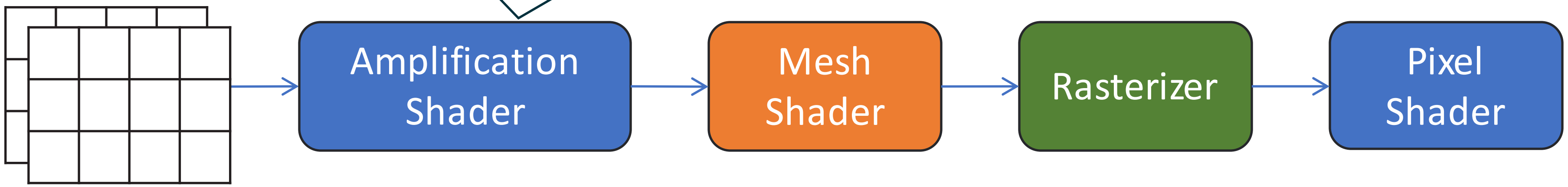
AMPLIFICATION SHADERS



AMPLIFICATION SHADERS

```
[NumThreads(64, 1, 1)]
void AmplificationShader(uint dtid : SV_DispatchThreadID) {
    ...

    DispatchMesh(x, y, z, payload);
}
```

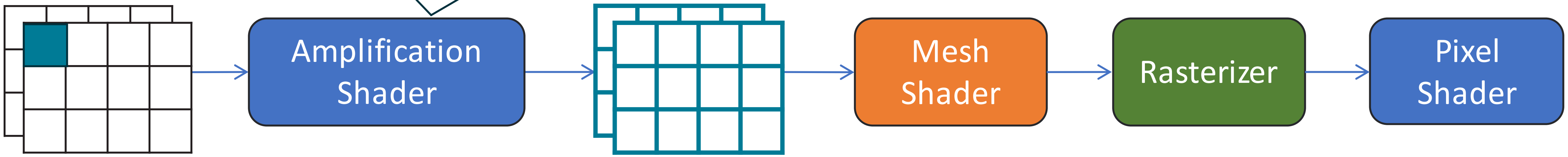
DispatchMesh(4, 3, 2)



AMPLIFICATION SHADERS

```
[NumThreads(64, 1, 1)]
void AmplificationShader(uint dtid : SV_DispatchThreadID) {
    ...

    DispatchMesh(x, y, z, payload);
}
```

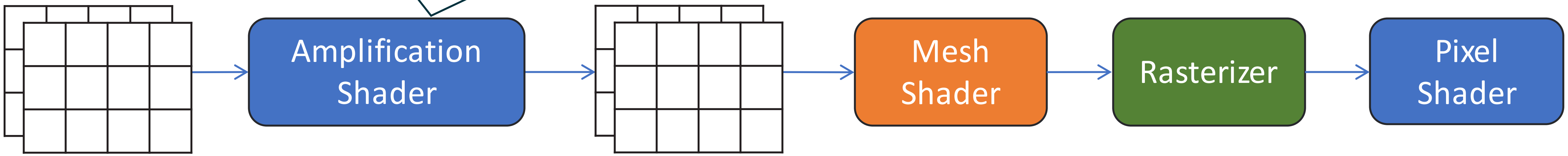
```
DispatchMesh(4, 3, 2)
```



AMPLIFICATION SHADERS – DYNAMIC LOD

```
[NumThreads(64, 1, 1)]
void AmplificationShader(uint dtid : SV_DispatchThreadID,
                         uint gtid : SV_GroupThreadID)
{
    uint lod, meshletCount;
    ComputeLevelOfDetail(instanceInfo[dtid], lod, meshletCount);

    payload.lod[gtid] = lod;

    DispatchMesh(WaveActiveSum(meshletCount), 1, 1, payload);
}
```

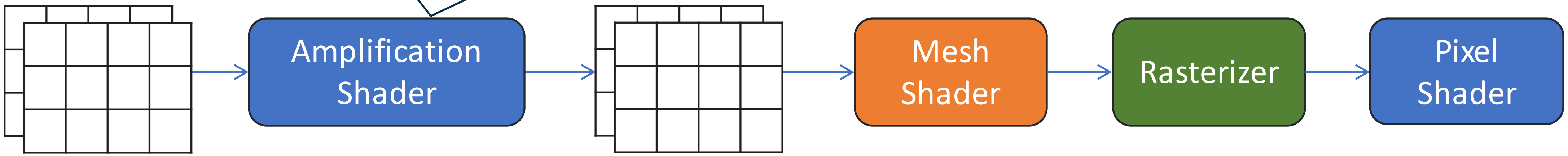
`DispatchMesh(4, 3, 2)`



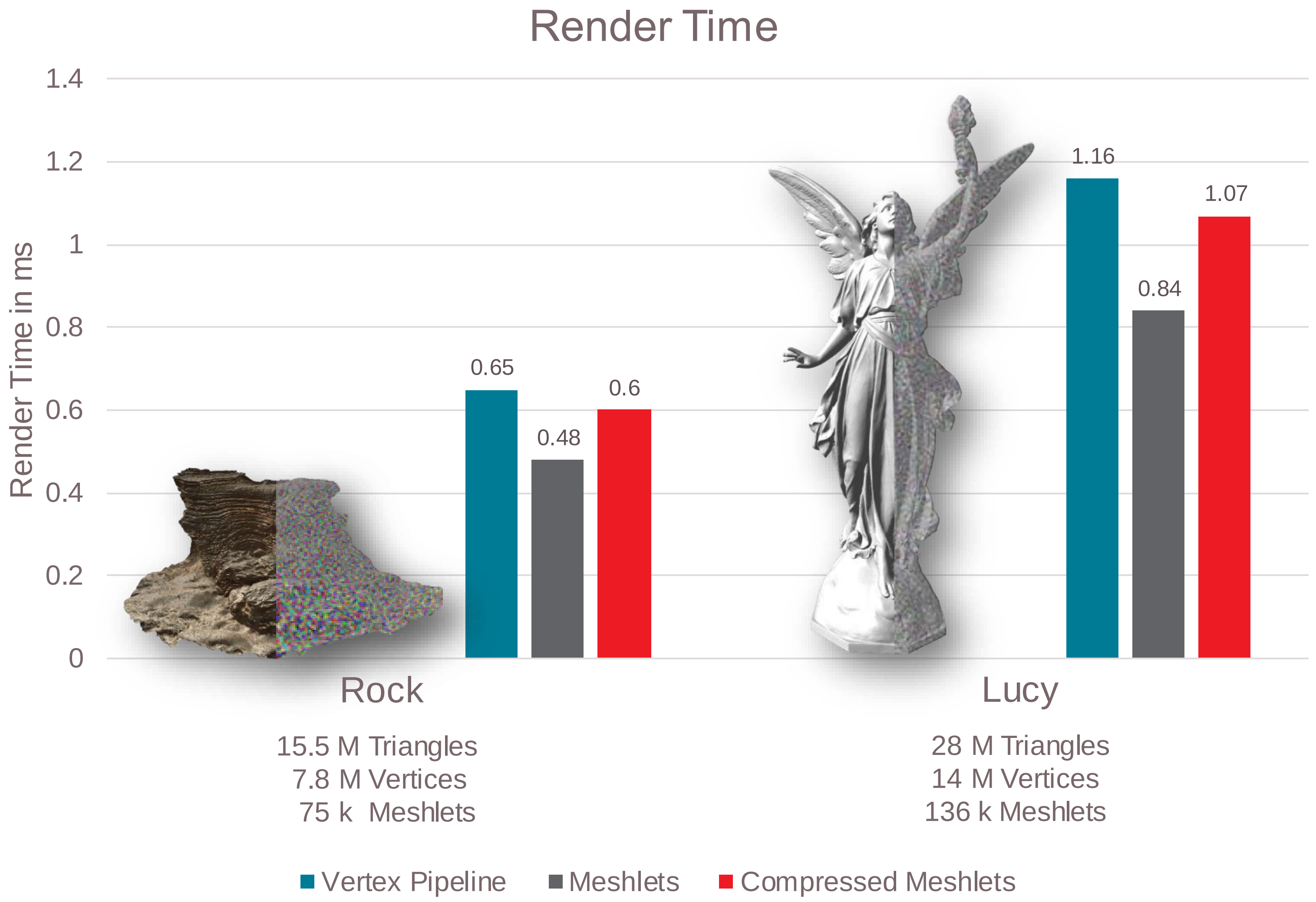
AMPLIFICATION SHADERS – CULLING

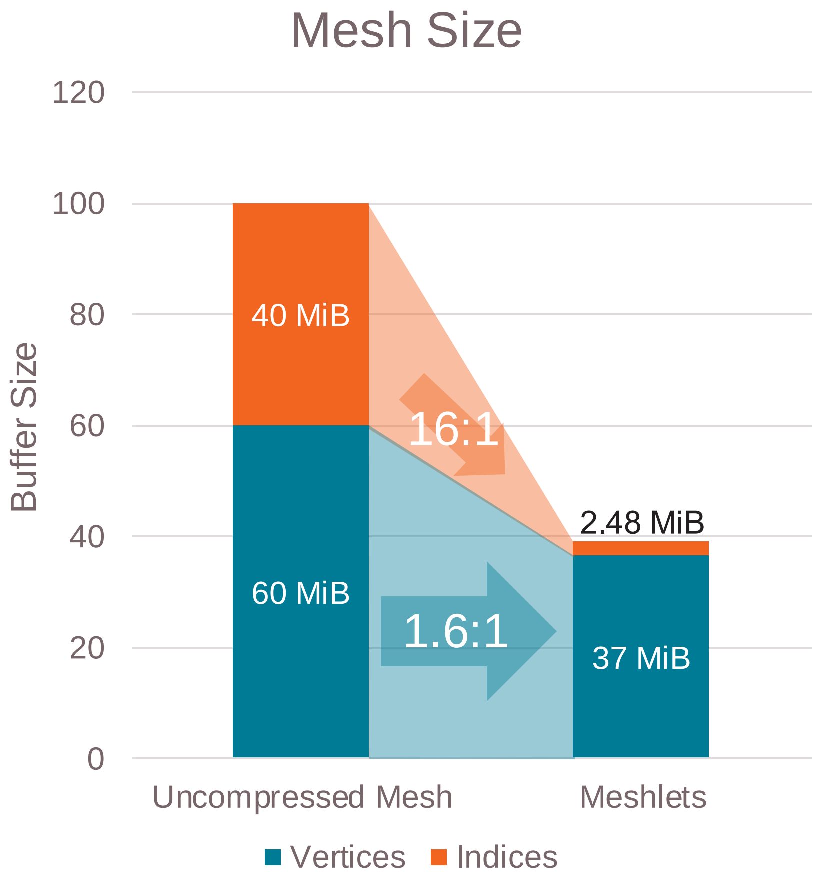
```
[NumThreads(64, 1, 1)]
void AmplificationShader(uint dtid : SV_DispatchThreadID)
{
    bool visible = IsMeshletVisible(Meshlets[dtid]);

    if (visible) {
        const uint index = WavePrefixCountBits(visible);
        payload.meshletIds[index] = dtid;
    }

    DispatchMesh(WaveActiveCountBits(visible), 1, 1, payload);
}
```

`DispatchMesh(4, 3, 2)`



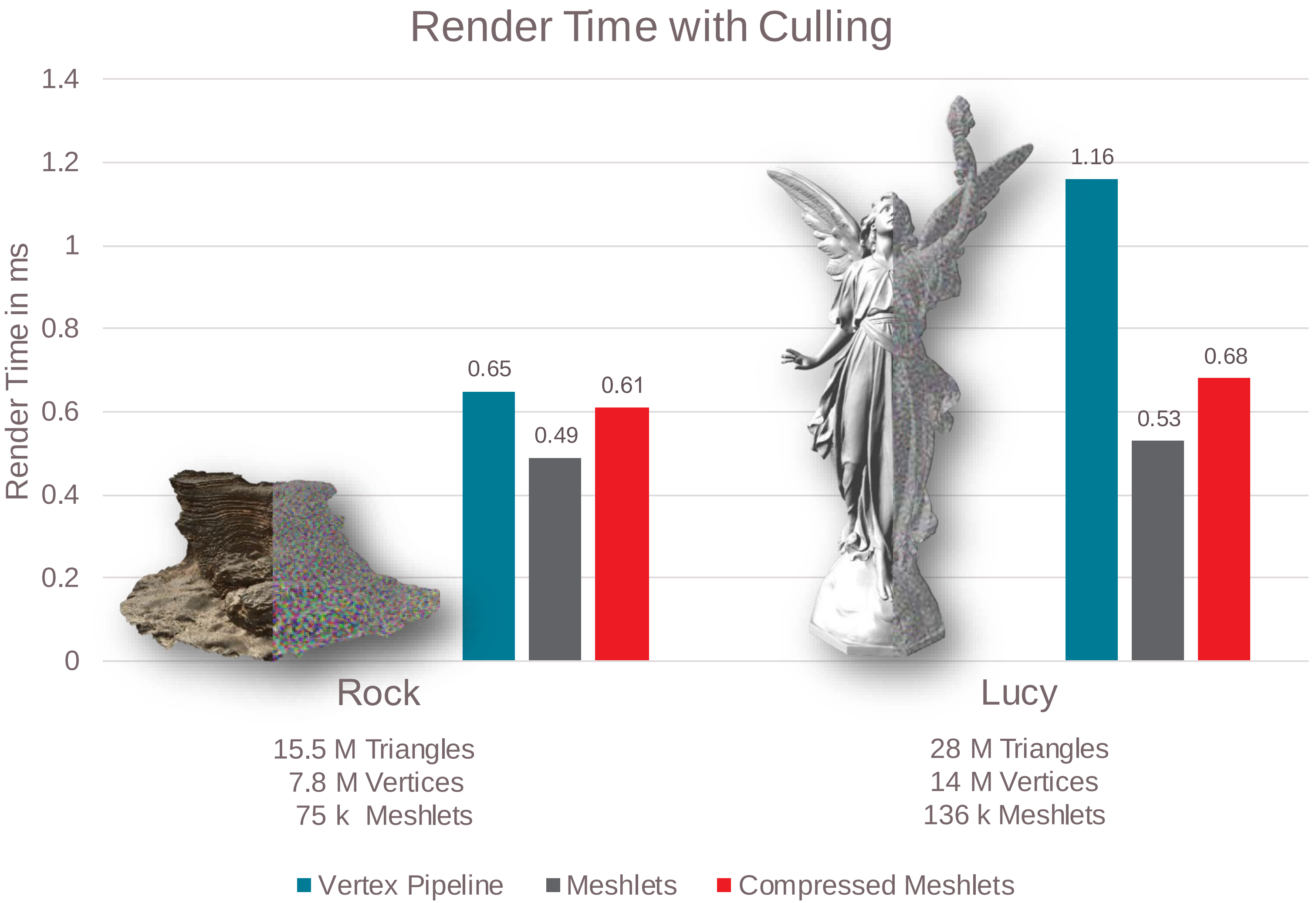
COMPRESSION

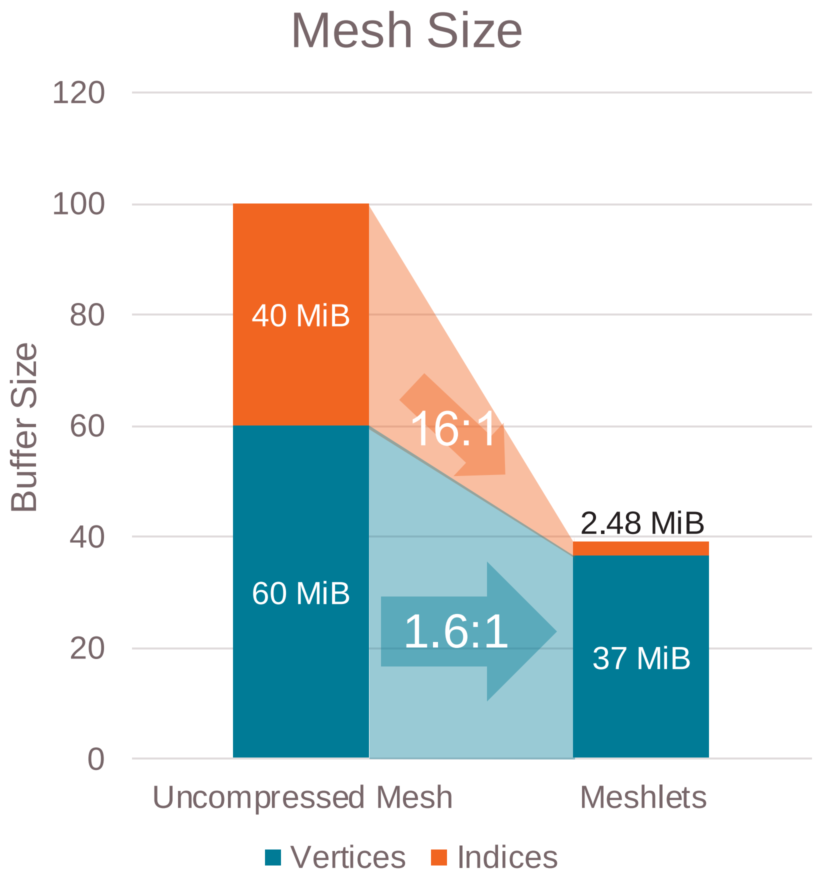




GPU: AMD Radeon™ RX 7900 XTX; driver version: 24.1.1; Meshes: Layered Rock by Aixterior, Lucy by Stanford Computer Graphics Laboratory

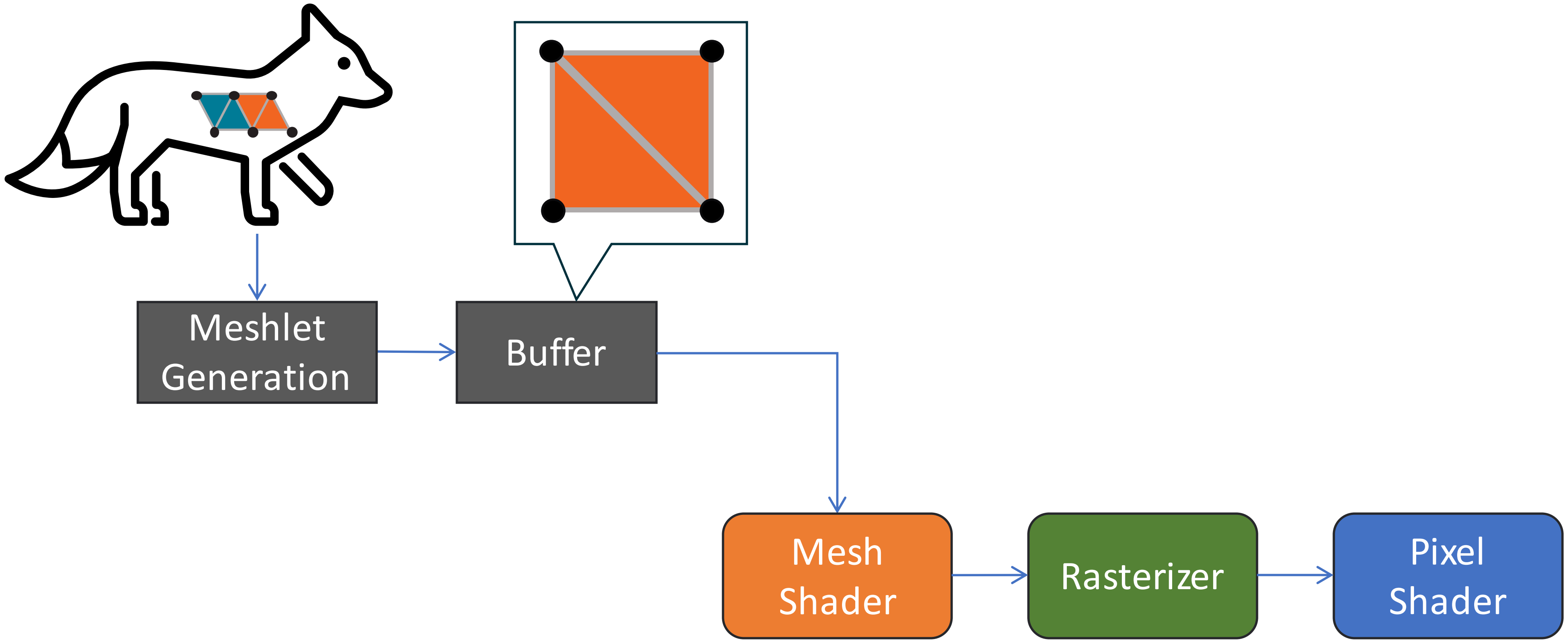
COMPRESSION



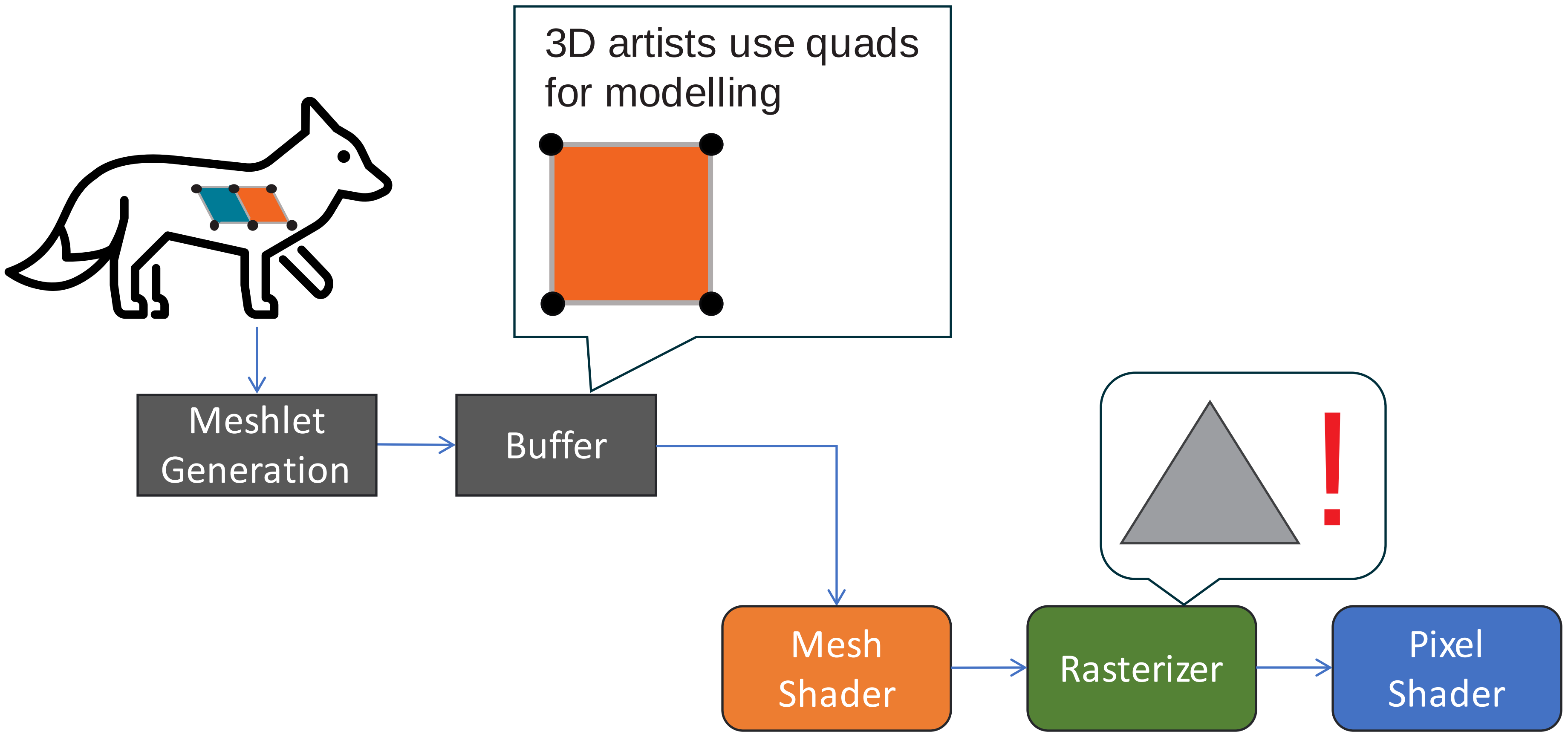


GPU: AMD Radeon™ RX 7900 XTX; driver version: 24.1.1; Meshes: Layered Rock by Aixterior, Lucy by Stanford Computer Graphics Laboratory

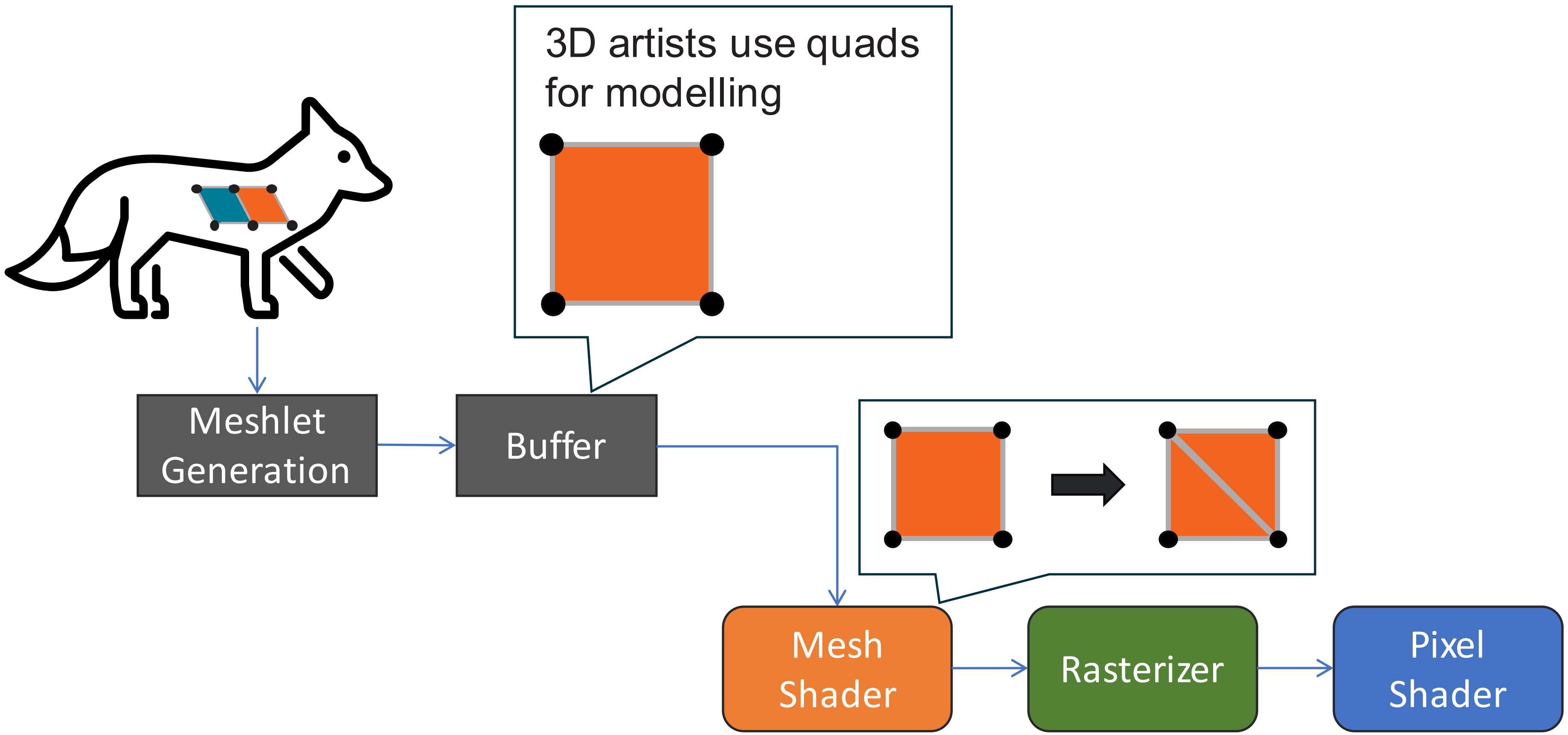
QUADRILATERAL PRIMITIVE RASTERIZATION



QUADRILATERAL PRIMITIVE RASTERIZATION

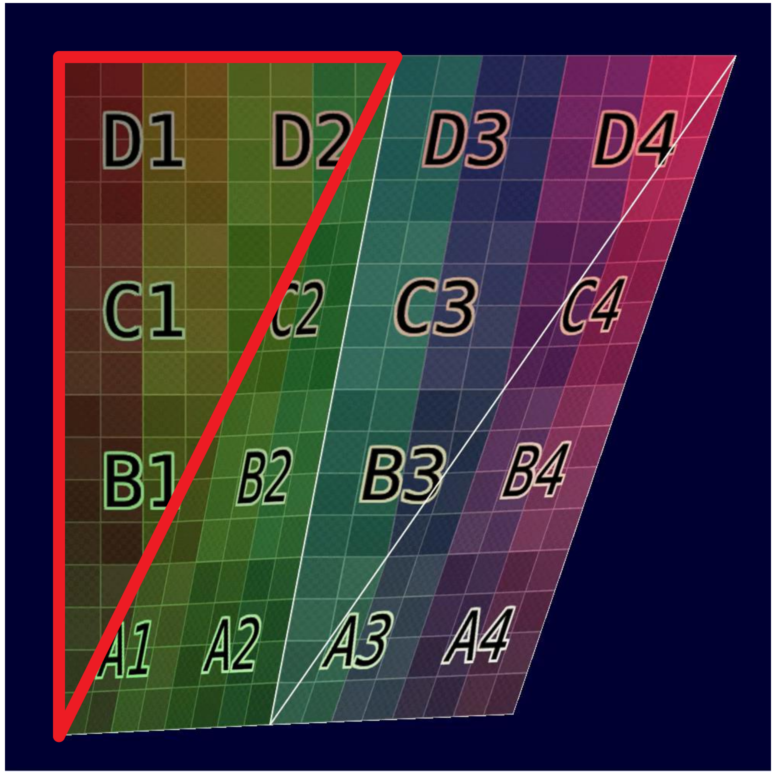


QUADRILATERAL PRIMITIVE RASTERIZATION

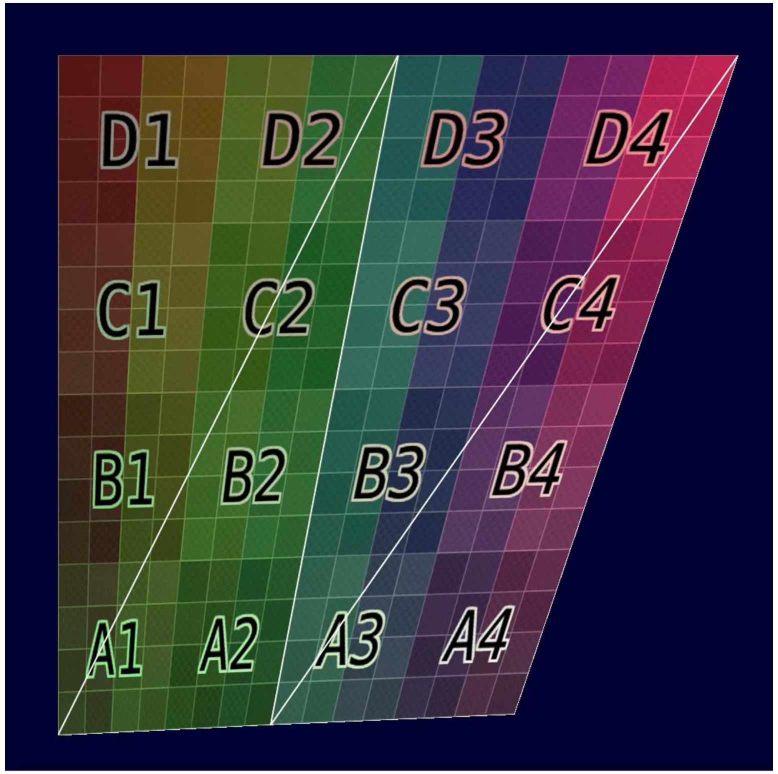


QUADRILATERAL PRIMITIVE RASTERIZATION

Barycentric Interpolation

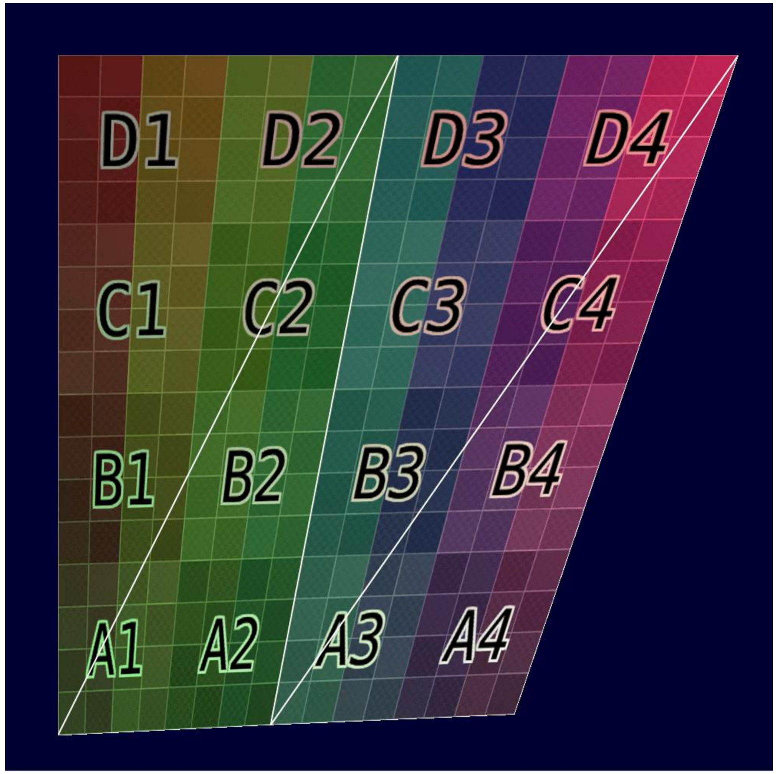


Bilinear Interpolation



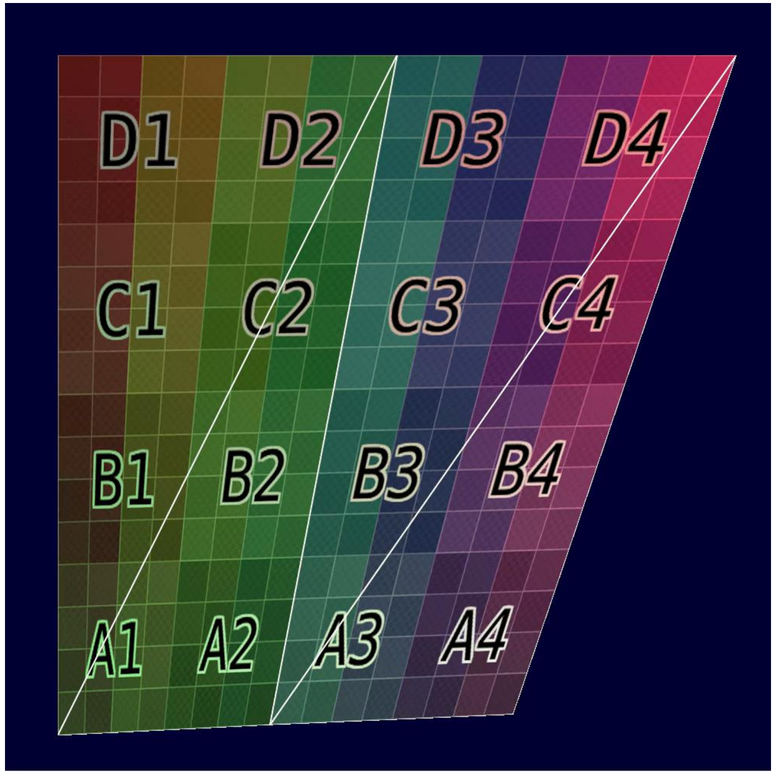
QUADRILATERAL PRIMITIVE RASTERIZATION

Bilinear Interpolation



QUADRILATERAL PRIMITIVE RASTERIZATION

Bilinear Interpolation

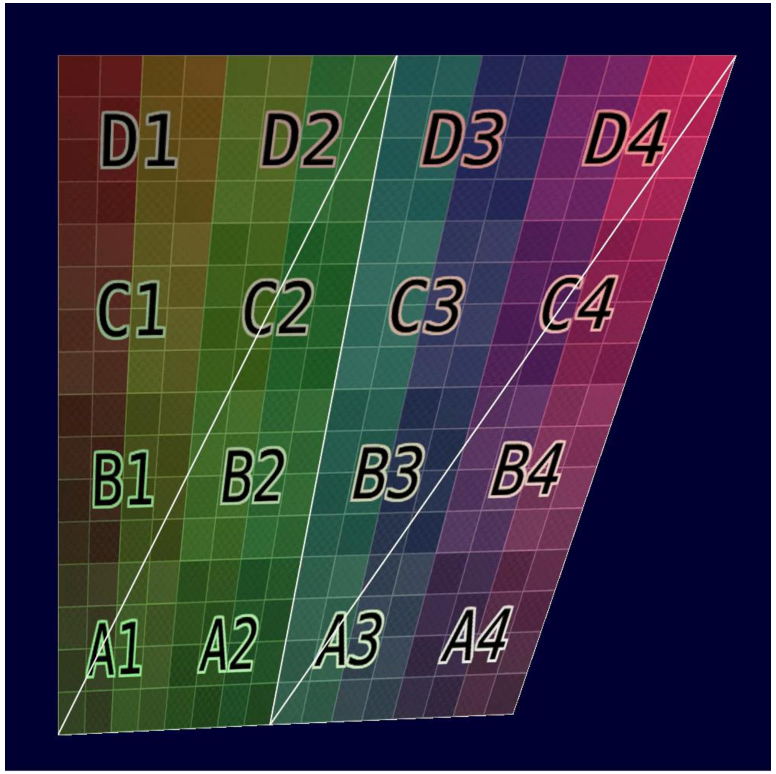


Solution: Primitive Attributes!

```
[NumThreads(128, 1, 1)]
[OutputTopology("triangle")]
void MeshShader(
    out indices uint3 tris[MAX_TRIANGLES],
    out primitives PrimitiveAttributes prims[MAX_TRIANGLES],
    out vertices VertexOutput verts[MAX_VERTICES],
```

QUADRILATERAL PRIMITIVE RASTERIZATION

Bilinear Interpolation

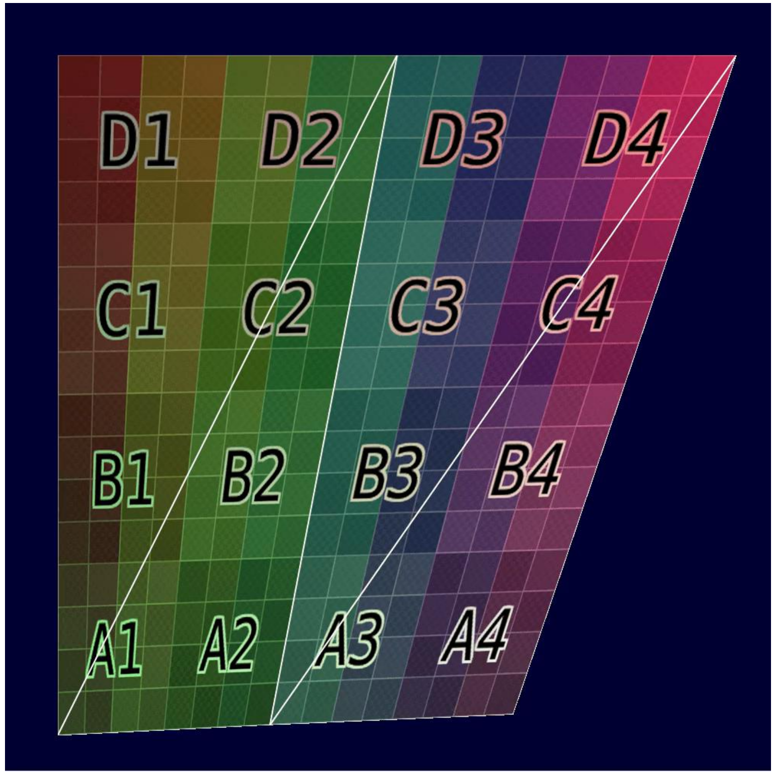


Implementing Bilinear Interpolation using the Mesh Shader approach has certain advantages:

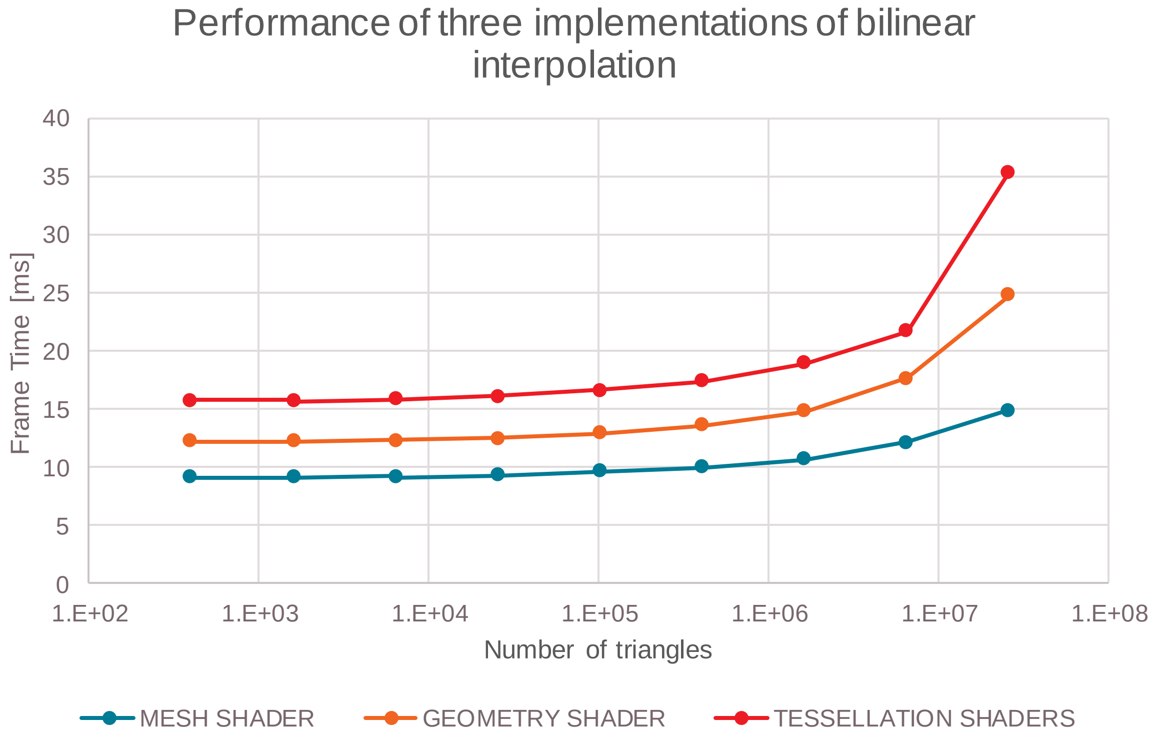
- Flexibility in how mesh data are passed to the graphics pipeline
  - We can create meshlets with quadrilateral primitives underneath
- Some calculations for the bilinear interpolation are done in Mesh Shader and some in Pixel Shader
  - Efficient data exchange between these two stages is important
  - Data can be shared per vertex but also per primitive

QUADRILATERAL PRIMITIVE RASTERIZATION

Bilinear Interpolation

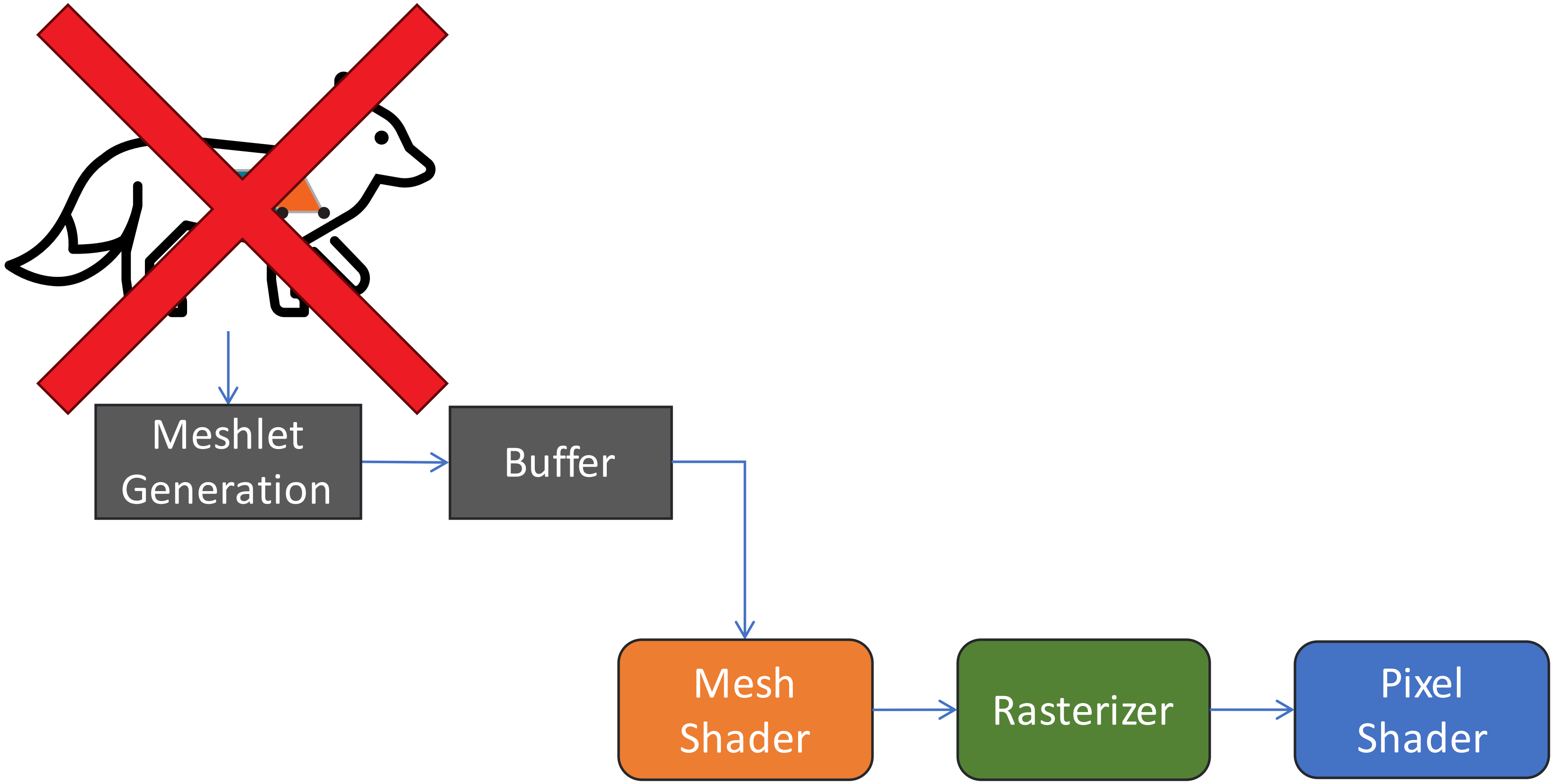


Faster than Tessellation/Geometry pipeline!



GPU: AMD Radeon™ RX 7900 XTX; driver version: 24.1.1

PROCEDURAL GEOMETRY



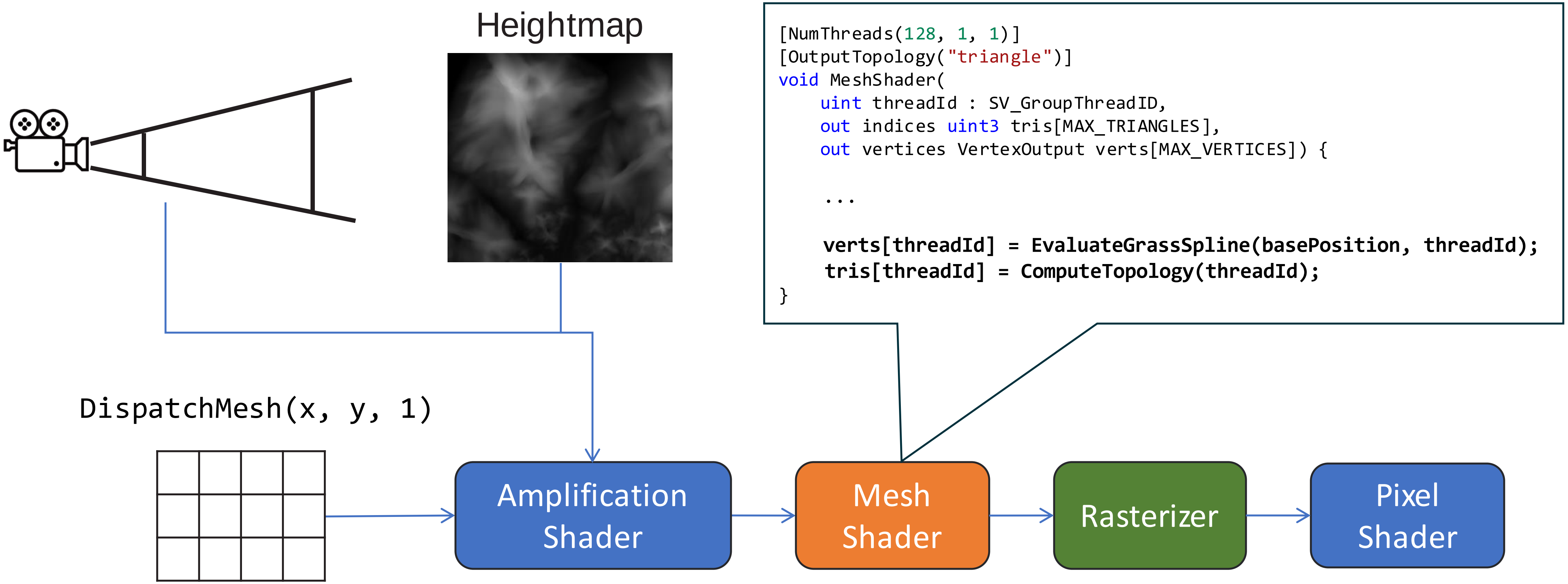
PROCEDURAL GEOMETRY

```
[NumThreads(128, 1, 1)]
[OutputTopology("triangle")]
void MeshShader(
    uint threadId : SV_GroupThreadID,
    out indices uint3 tris[MAX_TRIANGLES],
    out vertices VertexOutput verts[MAX_VERTICES]) {

    ...

    verts[threadId] = EvaluateGrassSpline(basePosition, threadId);
    tris[threadId] = ComputeTopology(threadId);
}
```

PROCEDURAL GEOMETRY

```
[NumThreads(128, 1, 1)]
[OutputTopology("triangle")]
void MeshShader(
    uint threadId : SV_GroupThreadID,
    out indices uint3 tris[MAX_TRIANGLES],
    out vertices VertexOutput verts[MAX_VERTICES]) {

    ...

    verts[threadId] = EvaluateGrassSpline(basePosition, threadId);
    tris[threadId] = ComputeTopology(threadId);
}
```

PROCEDURAL GEOMETRY




```
[NumThreads(128, 1, 1)]
[OutputTopology("triangle")]
void MeshShader(
    uint threadId : SV_GroupThreadID,
    out indices uint3 tris[MAX_TRIANGLES],
    out vertices VertexOutput verts[MAX_VERTICES]) {

    ...

    verts[threadId] = EvaluateGrassSpline(basePosition, threadId);
    tris[threadId] = ComputeTopology(threadId);
}
```


50+ million triangles
< 3 milliseconds

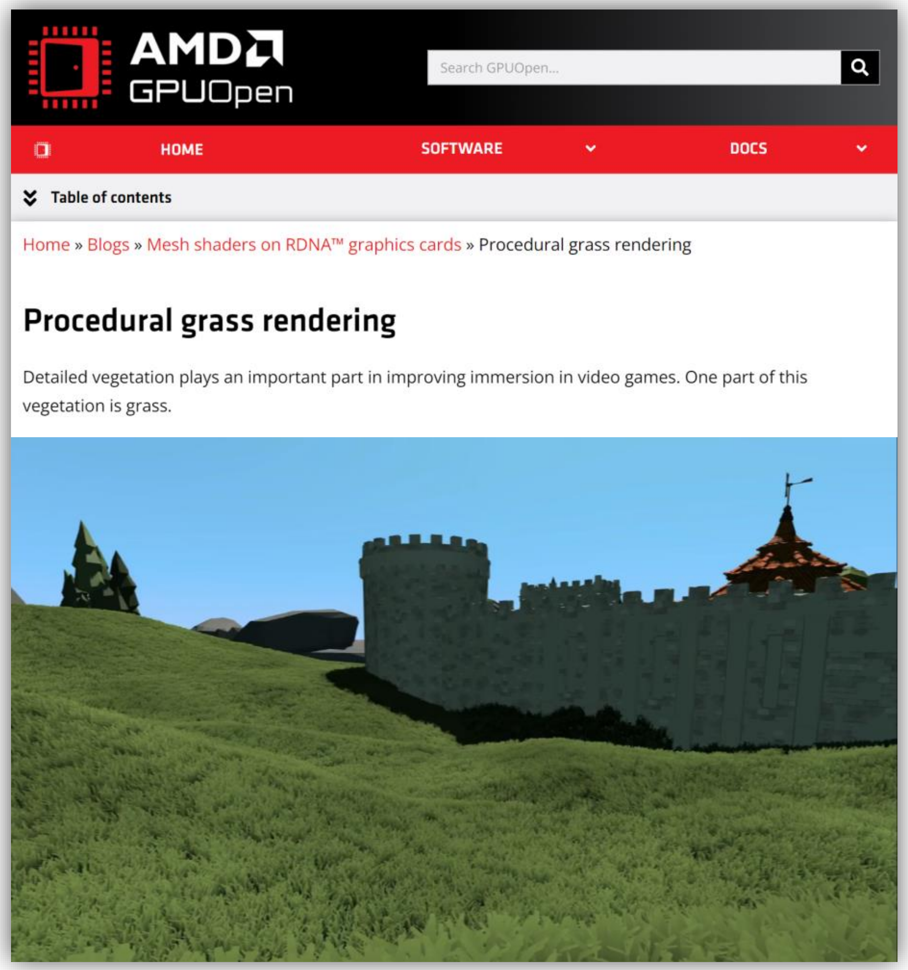
PROCEDURAL GEOMETRY

Learn more on
GPUOpen.com

Follow us on X
https://twitter.com/GPUOpen

Follow us on Mastodon
https://mastodon.gamedev.place/@gpuopen

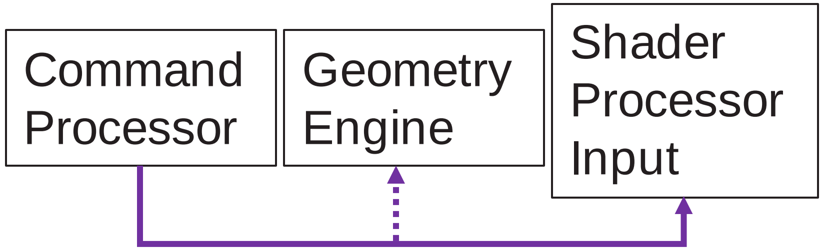
知乎 **Follow us on Zhihu**
https://www.zhihu.com/org/gpuopen-7



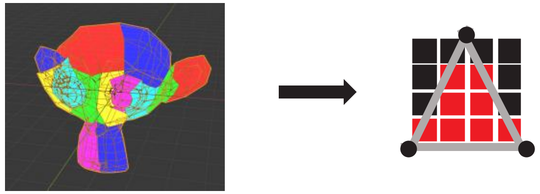
PERFORMANCE CONSIDERATIONS

PERFORMANCE CONSIDERATIONS - AGENDA

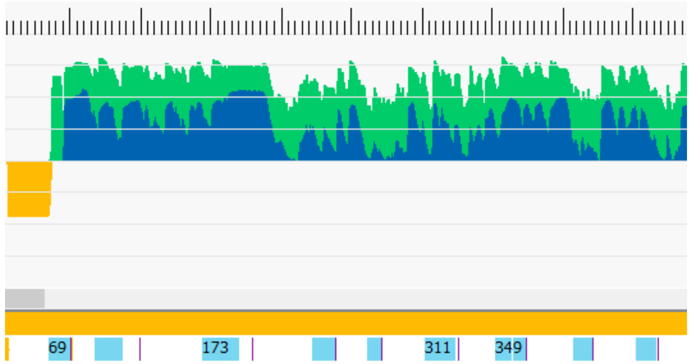
GEOMETRY ENGINE



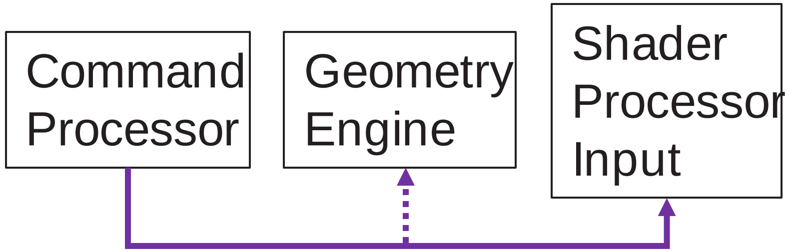
MESH SHADER



AMPLIFICATION (TASK) SHADER



GEOMETRY ENGINE – VERTEX REUSE



- The geometry engine holds a cache for vertex reuse
- Avoids re-shading vertices
- But depends on fixed input structure of vertex pipeline

- Asset pipelines can pack the vertices to make the best use of vertex reuse
- Mesh shaders would not take advantage of this and potentially re-shade vertices multiple times

VS: ☺	0	1	2	2	3	0	1	4	5	5	2	1	6	5	4	4	7	6	8 shaded, 10 re-used
MS: ☹	0	1	2	2	3	0	1	4	5	5	2	1	6	5	4	4	7	6	18 shaded

Mesh shader pipeline might run slower than traditional vertex/pixel shader pipeline!

GEOMETRY ENGINE – VERTEX REUSE

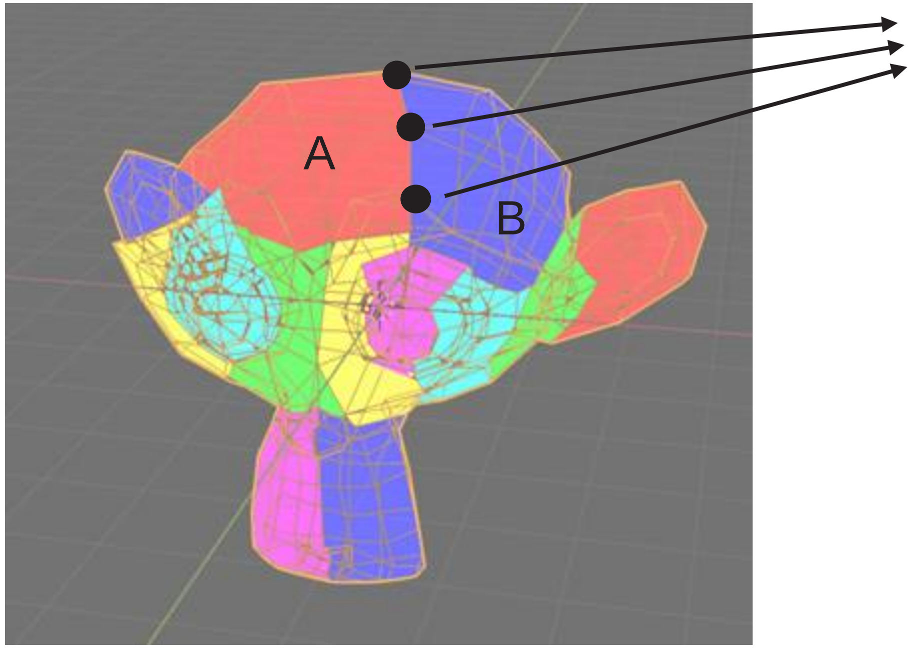
- Vertex reuse optimization has to be done during meshlet generation

VS Index Buffer:	0	1	2	2	3	0	1	4	5	5	2	1	6	5	4	4	7	6	8 shaded vertices
Meshlet Index Buffer:	0	1	2	2	3	0	1	4	5	5	2	1	6	5	4	4	7	6	6 primitives
Meshlet Unique Vertices Buffer:	0	1	2	3	4	5	6	7											8 vertices

- MS thread reads directly from the buffers
- MS thread group size is max 128 (DirectX® 12)
- MS max output: 256 vertices and 256 primitives (DirectX® 12)

- The fixed function vertex reuse cache is typically only for 32 vertices

GEOMETRY ENGINE – VERTEX REUSE



Duplicated vertices
Part of Meshlet A and Meshlet B

Border vertices might need to be duplicated since they fall into multiple meshlets

→ Mesh Shader potentially need to process less vertices in total compared to VS

MESHLET GENERATION – PART 1

- There are different metrics in how meshlets can be constructed
- Depending on the problem case, one or the other is more suitable (content-specific!)

Common metrics that can be considered are

- Number of border vertices
  - Have to be duplicated
- Size of bounding box
  - Helpful for culling
  - Quantization precision
- Triangle strips (topologically connected triangles)
  - Can help with compression rate
  - Might lower performance

- Meshlets can be a collection of loose vertices/primitives

- There’s still lots of room for research in this area

GEOMETRY ENGINE

The GE for Vertex Shaders

- Determines vertexID for each vertex
- Vertex re-use cache
- Prepares the shader export
- Initiate launch of shaders

The GE for Mesh Shaders

- Determines threadID for each thread
- Prepares the shader export
- Initiate launch of shaders

→ Launch rate for mesh shader is faster

GEOMETRY ENGINE

- Prepares the shader export
  - Allocates enough space for the maximum number of exported vertices and primitives per thread group

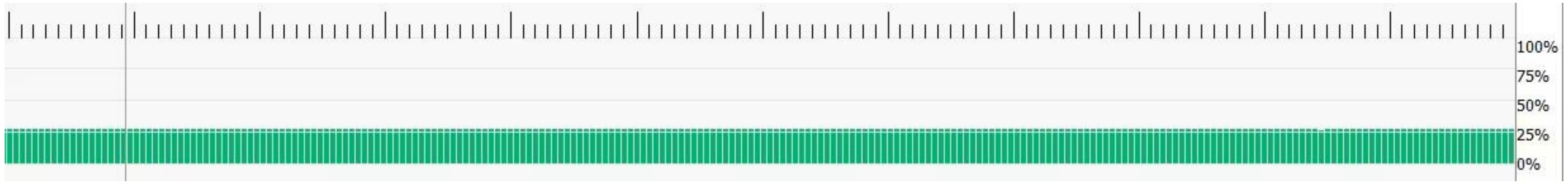
```
[NumThreads(128, 1, 1)]
[OutputTopology("triangle")]
void MeshShader(
    out indices uint3 tris[MAX_TRIANGLES],
    out vertices VertexOutput verts[MAX_VERTICES],
```

Allocates enough space for `MAX_TRIANGLES` primitives and `MAX_VERTICES` vertices

  - If the export buffer is full, no new waves can be launched
  - Can limit the max. occupancy of mesh shaders
- Space in the shader export is finite
- Designed for “average” mesh shader workloads to reach rasterizer triangle throughput

GEOMETRY ENGINE - OCCUPANCY

- An occupancy of ~25% can be enough to reach the triangle throughput limit



In AMD Radeon™ GPU Profiler (RGP) colored as geometry shader

- Complex mesh shaders might see higher occupancy (~50%)
- Tend to get limited by available memory in the shader export
- `MAX_TRIANGLES` and `MAX_VERTICES` should be set as low as possible

- Mesh shader occupancy can also be limited by
  - VGPRs
  - Group shared memory (LDS)
  - Launch rate

MESH SHADER – TRIANGLE THROUGHPUT

An indicator for a mesh shader being limited by rasterizer throughput is a high export instruction latency, particularly on the first exp instruction

This can be inspected in RGP under the instruction timing tab

MESH SHADER

- Work a lot like a compute shader

- [numthreads(x,y,z)] to define thread group size
- ThreadID used to read from the input buffer and write to the output buffer
- These are shared buffers
- Any thread can read and write from and to any index

```
[NumThreads(128, 1, 1)]
[OutputTopology("triangle")]
void MeshShader(
    uint threadId : SV_GroupThreadID,
    out indices uint3 tris[MAX_TRIANGLES],
    out vertices VertexOutput verts[MAX_VERTICES]) {
    ...

    SetMeshOutputCounts(meshlet.vertex_count, meshlet.triangle_count);
    if (threadId < meshlet.vertex_count)
        verts[threadId] = ProcessVertex(vertex_buffer[threadId]);
    if (threadId < meshlet.triangle_count)
        tris[threadId] = ProcessPrimitive(triangle_buffer[threadId])
}
```

- Unlike a compute shader, the vertices and primitives are exported to the shader export

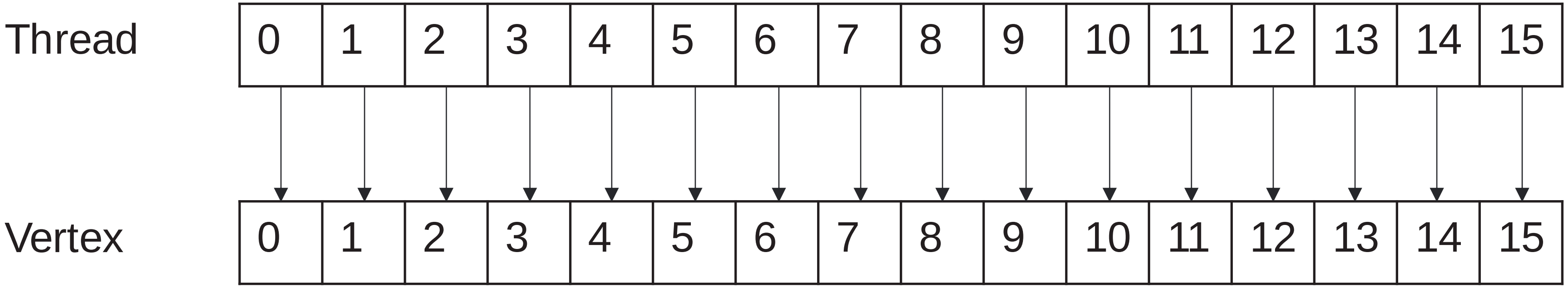
```
exp         prim v3, off, off, off done row_en
```

```
exp         pos0 v24, v23, v25, v12 done row_en
```

- That‘s what the geometry engine allocates space for ☺
- Vertex attributes are exported to memory via a regular buffer store (AMD RDNA™ 3)
  - On AMD RDNA™ 2 it‘s via exp param

MESH SHADER - EXPORT

- On AMD RDNA™, the order of primitives and vertices in the shader export is defined by the order of threads in the thread group
- This means thread n exports vertex n and primitive n
  - This is how vertex shaders export their vertices!

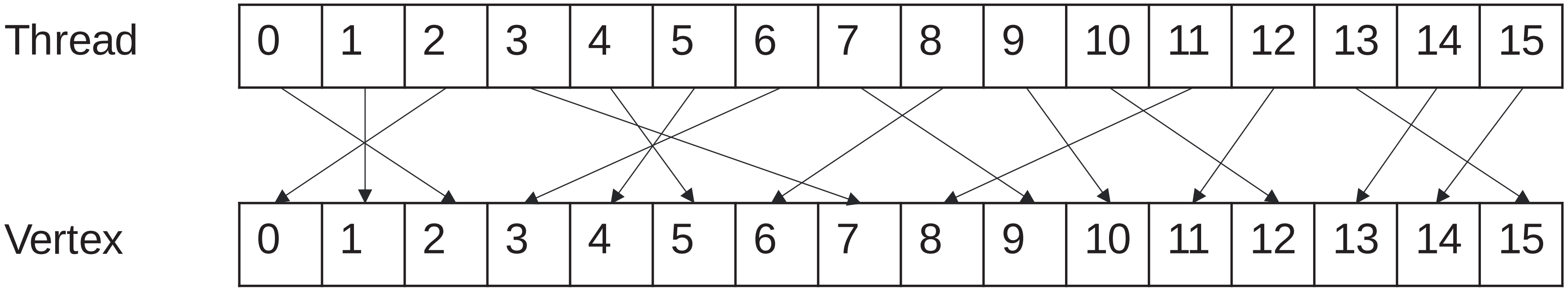



```
if (threadId < meshlet.vertex_count)
    verts[threadId] = ProcessVertex(vertex_buffer[threadId]);
if (threadId < meshlet.triangle_count)
    tris[threadId] = ProcessPrimitive(triangle_buffer[threadId])
```

- Unfortunately, this clashes with the flexibility of mesh shaders
  - Any thread can write to any index in the output buffers

MESH SHADER - EXPORT

- Mesh shaders can do something like this:

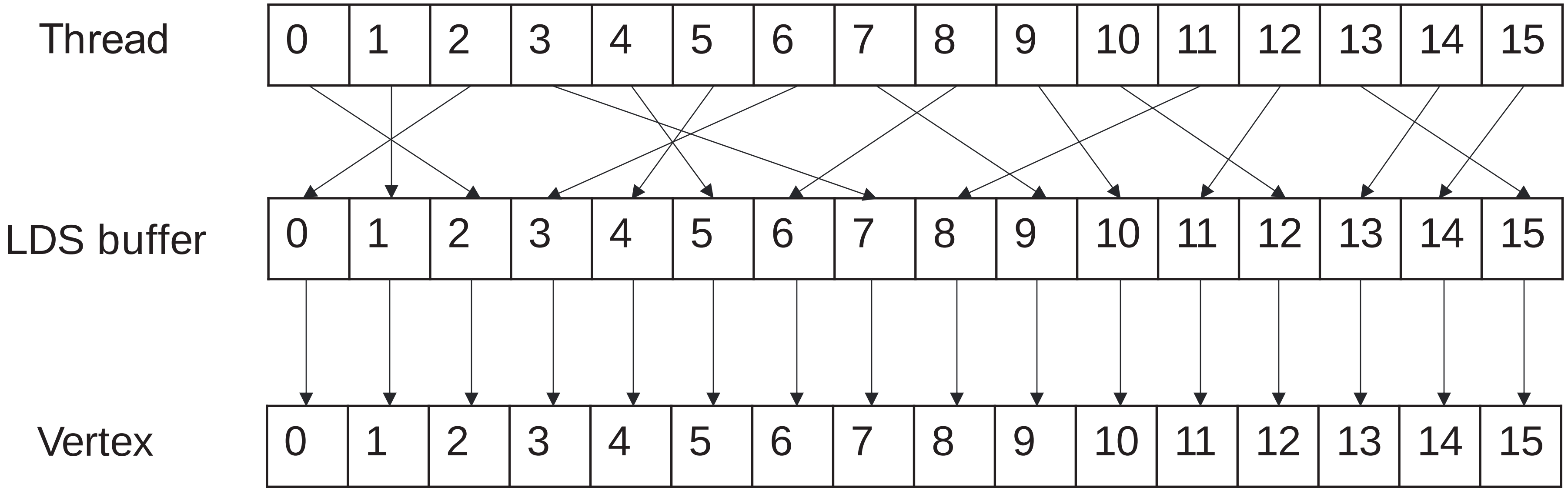


- This does not comply with the order of primitives and vertices in the shader export

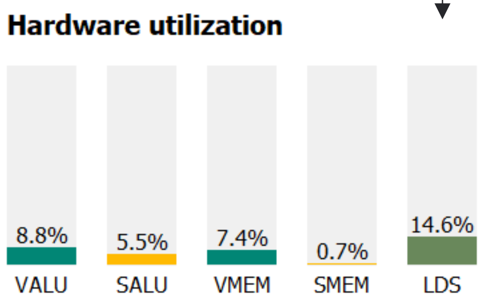
- We need to fix this somehow

MESH SHADER - EXPORT

- Mesh shaders can do something like this:



LDS utilization increases



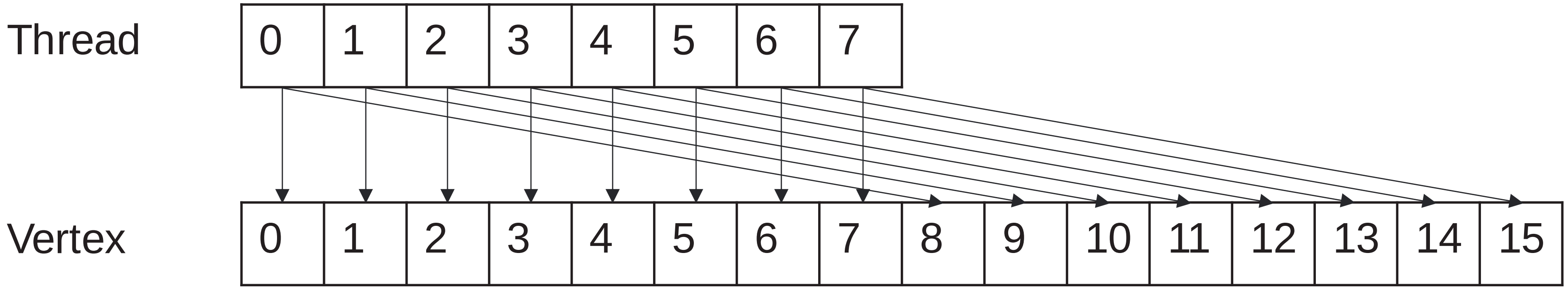
- Group shared memory (LDS) is used to fix this!

- Can be seen in the ISA:

```
ds_load_b32        v0, v0
s_mov_b32          m0, s3
s_waitcnt          lgkmcnt(0)
exp                prim v0, off, off, off done row_en
```

MESH SHADER - EXPORT

- A thread can also export multiple vertices and primitives

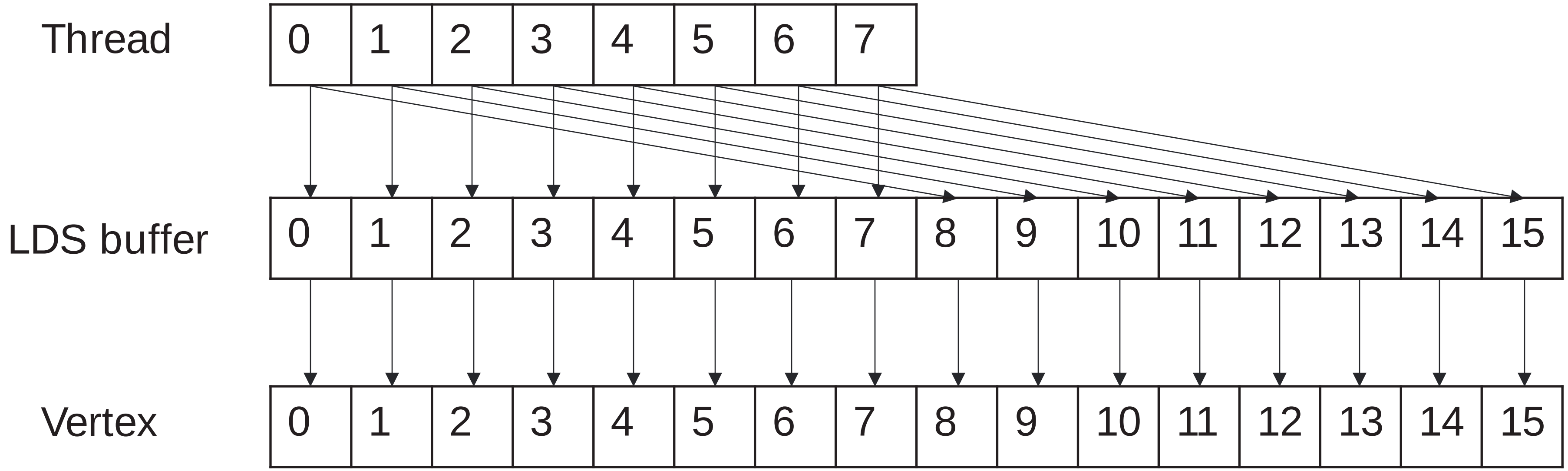



```
Vertices[thread_id.x] = UnpackVertex(meshlet, thread_id.x);
Vertices[thread_id.x + 8] = UnpackVertex(meshlet, thread_id.x + 8);
```

- **On AMD RDNA™ 3**, this is achieved via a **wave-wide offset** to the export instruction

MESH SHADER - EXPORT

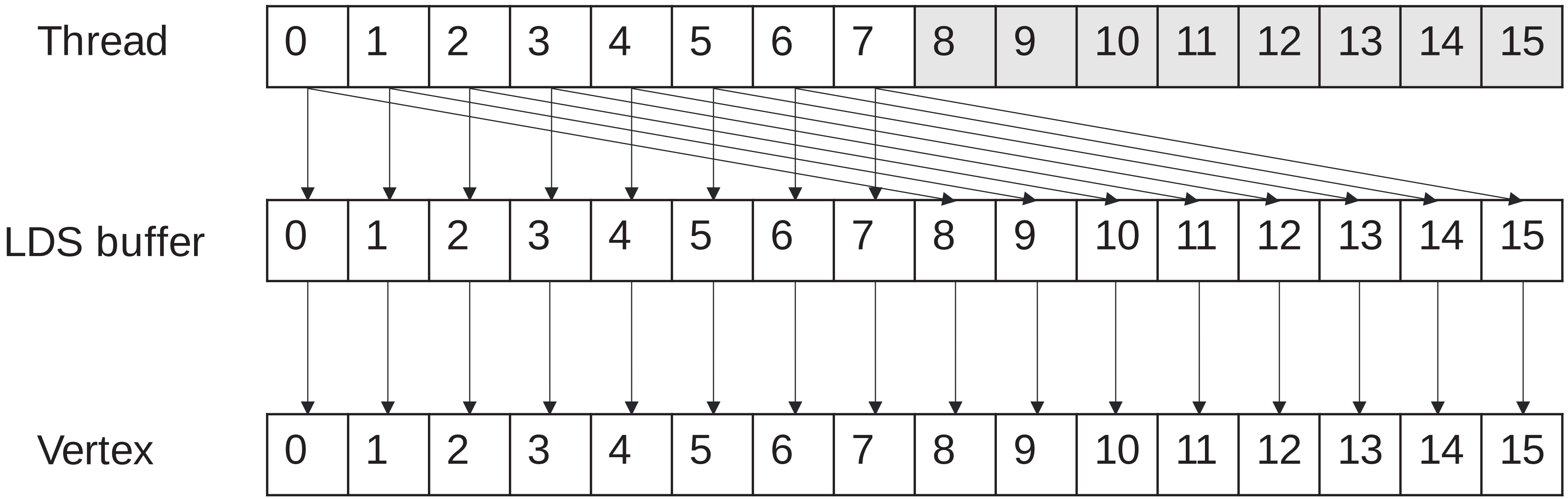
- A thread can also export multiple vertices and primitives



- **On AMD RDNA™ 3**, this is achieved via a **wave-wide offset** to the export instruction
- The compiler may choose to use group shared memory as a staging buffer

MESH SHADER – EXPORT AMD RDNA™ 2

- A small note for AMD RDNA™ 2: there is no wave-wide offset
- A thread can only export 1 vertex and 1 primitive max



- Shadow threads are launched that only export

MESH SHADER – EXPORT CONCLUSION

- Thread n should export vertex n and primitive n
- If multiple vertices and primitives are exported per thread, use a wave-wide offset

- Otherwise:
  - Latency increases → can decrease triangle throughput
  - Group shared memory usage increases

- Group shared memory can also be used explicitly in the HLSL shader
  - Efficient data exchange between threads within a thread group is possible

- Group shared memory is limited!

→ Can affect both, mesh shader and pixel shader occupancy

- If not enough group shared memory is available, scratch memory is used

MESHLET GENERATION - PART 2

- What is a recommended configuration?
- There are 2 quite common configurations:

- V = 128, T = 256
- Thread group size = 128

→ Larger meshlets

- V = 64, T = 126 (or 128)
- Thread group size = 64

→ Smaller meshlets

MESHLET GENERATION - PART 2

- What is a recommended configuration?
- There are 2 quite common configurations:

- V = 128, T = 256 →
- Thread group size = 128

→ Larger meshlets

- V = 64, T = 126 (or 128) →
- Thread group size = 64

→ Smaller meshlets

More primitives than vertices

Main compute workload is typically vertex transformations

The number of primitives should not be the limiting factor, but the number of vertices

MESHLET GENERATION - PART 2

- What is a recommended configuration?
- There are 2 quite common configurations:

- V = 128, T = 256
- Thread group size = 128

→ Larger meshlets

- V = 64, T = 126 (or 128)
- Thread group size = 64

→ Smaller meshlets

Thread group size = maximum number of vertices

Main compute workload is typically vertex transformations

Ensures 1 thread processes 1 vertex

If this is not possible,
try to divide the maximum number of vertices evenly across the threads

MESHLET GENERATION - PART 2

- What is a recommended configuration?
- There are 2 quite common configurations:

- V = 128, T = 256
- Thread group size = 128 → Larger meshlets have less border vertices in total

→ Larger meshlets

- V = 64, T = 126 (or 128)
- Thread group size = 64 → More border vertices
  A single thread group needs less resources
  If occupancy is limited by other pixel shader or other dispatches running in parallel, a smaller thread group can improve performance

→ Smaller meshlets

MESHLET GENERATION - PART 2

- What is a recommended configuration?
- There are 2 quite common configurations:

Bonus question: How should you export the primitives?

- V = 128, T = 256
- Thread group size = **128**

→ Larger meshlets

Use a thread-group-sided stride:

```
triangles[threadID] = ProcessPrimitive[threadID];
triangles[threadID + 128] = ProcessPrimitive[threadID + 128];
```

- V = 64, T = 126 (or 128)
- Thread group size = **64**

→ Smaller meshlets

```
triangles[threadID] = ProcessPrimitive[threadID];
triangles[threadID + 64] = ProcessPrimitive[threadID + 64];
```

MESH SHADER - CULLING

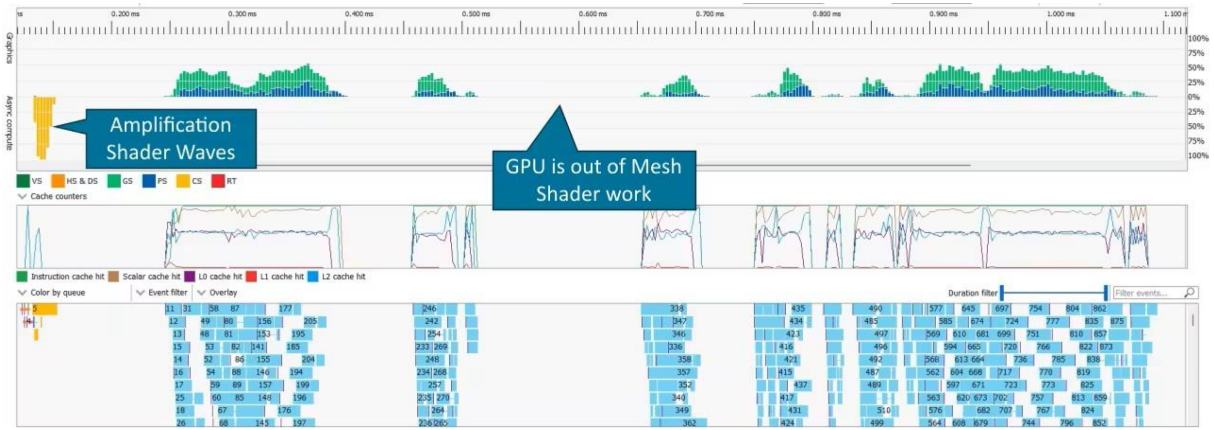
- Mesh shader can do triangle/primitive culling
- Might help if the fixed function cull rate is a bottleneck (rasterizer triangle throughput limited)

Remove manually from export	SV_CullPrimitive
• Done before `SetMeshOutputCounts` • No space for vertex attributes is allocated • Need to fix export indices via a wave-wide scan	• No effect on `SetMeshOutputCounts` • Space for vertex attributes is still allocated • No need to fix any export indices

- Export space is always allocated for the maximum # of vertices and primitives regardless of `SetMeshOutputCounts`
- In our experiments, SV_CullPrimitive was more beneficial
- The other option might be useful if there are very fat vertices
  - Where applicable, per-primitive attributes could already help

AMPLIFICATION (TASK) SHADER

- Amplification shaders add latency to the render process
- MS calls are executed in the same order as the amplification shader thread groups were launched
- Required to comply with the specified rasterization order

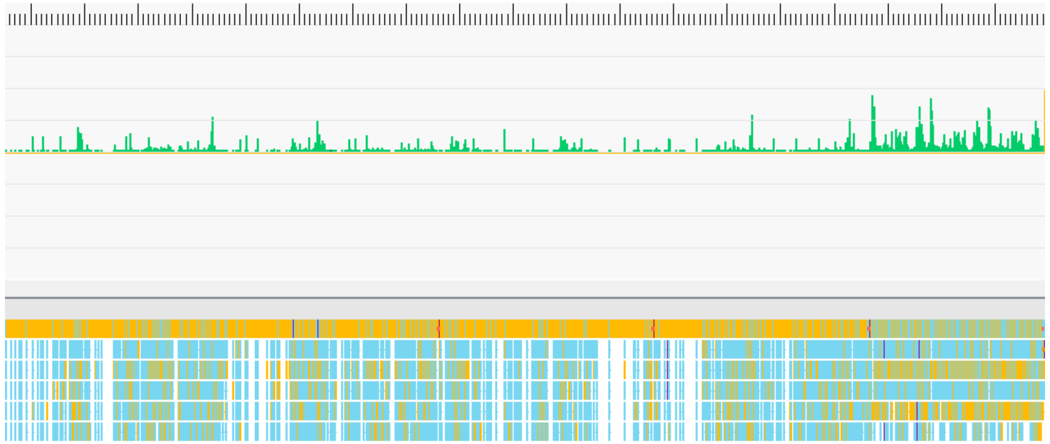


- AS thread groups need to launch enough MS thread groups to hide the latency
- Otherwise, there will be gaps and poor thread utilization

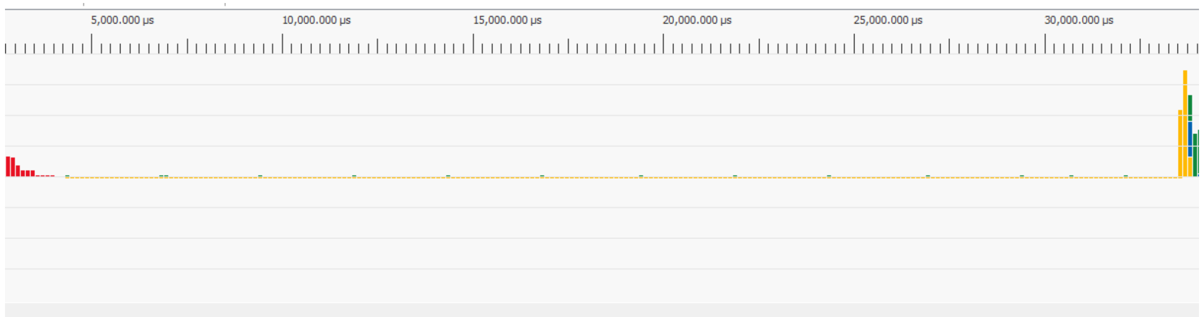
- Typically, AS thread groups should launch at least 32 MS thread groups
- For culling, try to process at least 32 or 64 meshlets per AS thread group

AMPLIFICATION SHADER – DYNAMIC LOD SELECTION

- An amplification shader is called per mesh to select dynamically the LOD and possibly cull



- Issues in above trace:
  - Each mesh only spawns a few MS draws if any (e.g., one AS thread group with only few threads active)
  - A lot of AS thread groups doing very little work and producing very little work
  - Overhead can be estimated by changing AS code to cull all meshlets:



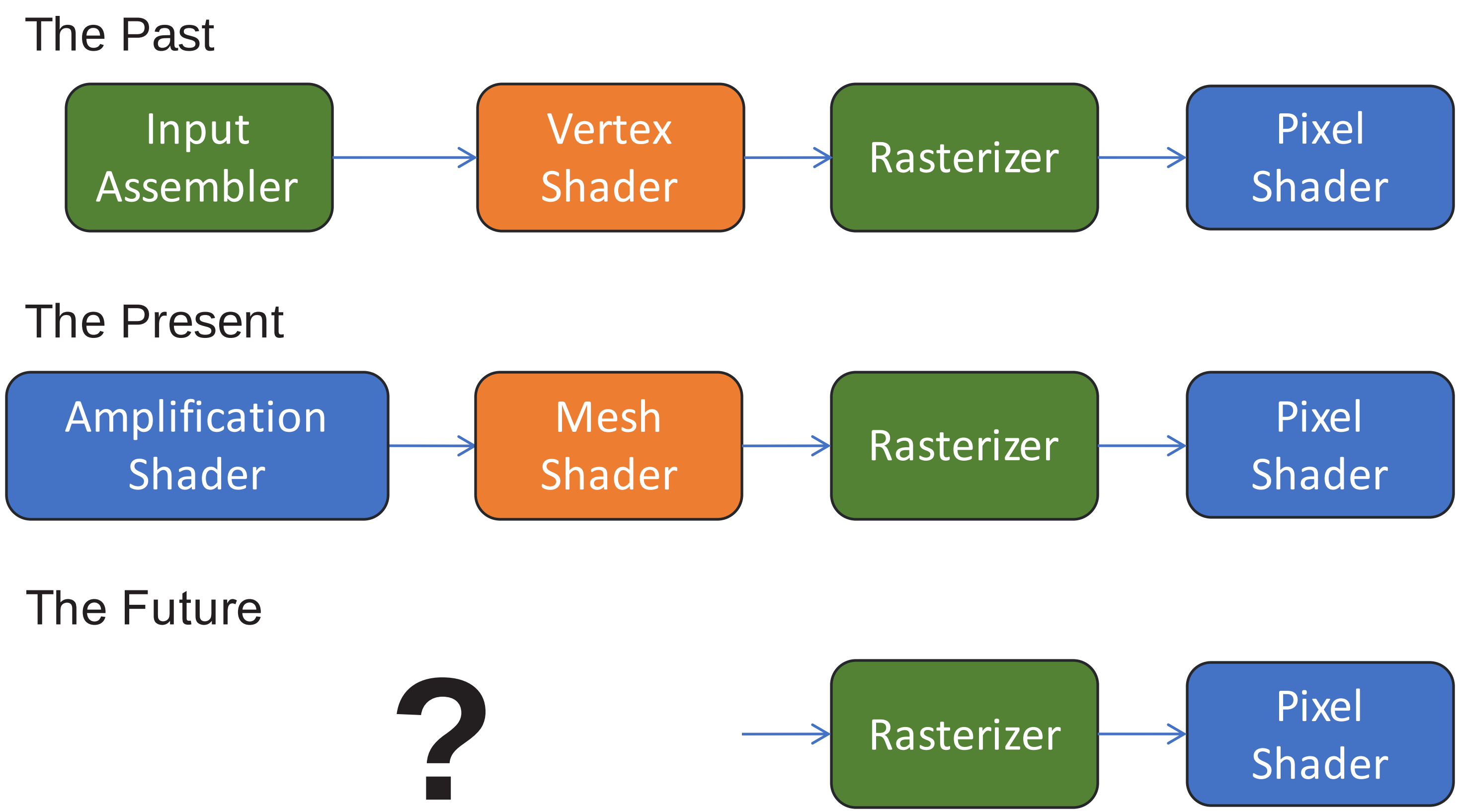
AMPLIFICATION SHADER - PAYLOAD

- Payload is stored in group shared memory
- Every thread in the AS thread group can read and write from it
- After the AS thread group finishes, the payload is copied to a ring buffer
  - This copy can be quite slow, every thread copies the entire payload (up to 16KiB)
- Also requires a lot of VGPRs
  - Load from group shared memory into VGPRs
  - Copy from VGPRs to ring buffer using buffer_store instruction

→ Use as little payload as possible!

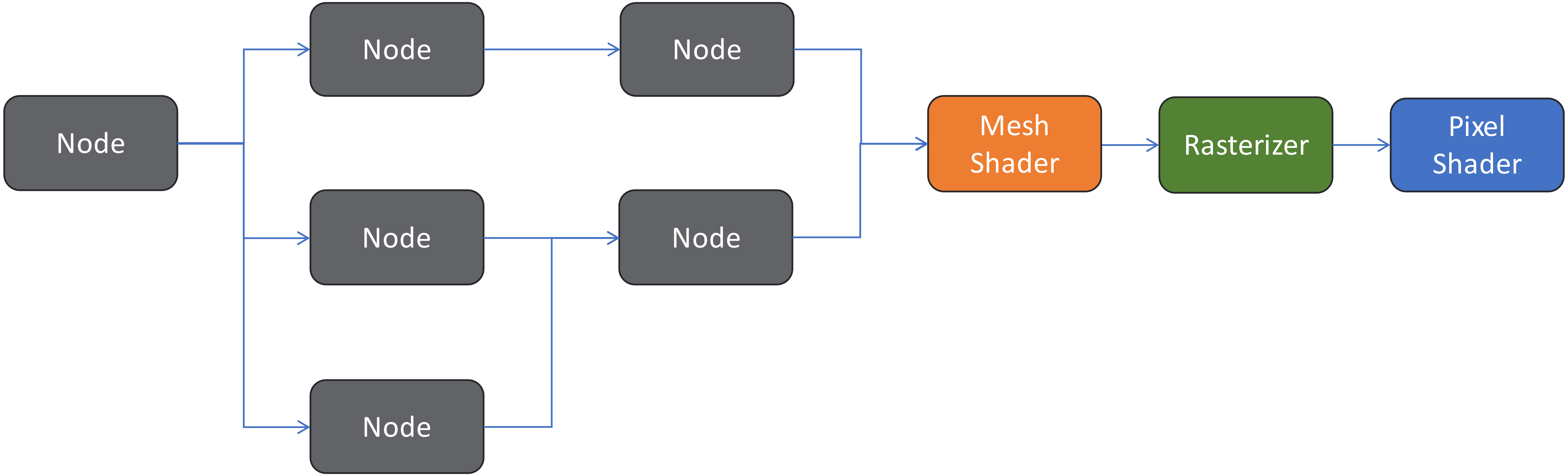
```
s_mov_b32          exec_lo, s1
v_mov_b32_e32      v0, 0
s_mov_b32          s1, s7
s_mul_i32          s2, s18, s13
s_add_i32          s2, s2, s17
s_load_b128        s[4:7], s[0:1], 0xd0
s_mul_i32          s2, s2, s12
s_delay_alu        instid0(SALU_CYCLE_1) | instskip(NEXT) | instid1(SALU_CYCLE_1)
s_add_i32          s2, s2, s16
s_add_i32          s2, s15, s2
ds_load_b128       v[4:7], v0
ds_load_b128       v[16:19], v0 offset:16
ds_load_b128       v[8:11], v0 offset:32
ds_load_b128       v[20:23], v0 offset:48
ds_load_b128       v[24:27], v0 offset:64
ds_load_b128       v[12:15], v0 offset:80
ds_load_b128       v[28:31], v0 offset:96
ds_load_b128       v[36:39], v0 offset:112
ds_load_b128       v[40:43], v0 offset:128
ds_load_b128       v[32:35], v0 offset:144
ds_load_b128       v[44:47], v0 offset:160
ds_load_b128       v[52:55], v0 offset:176
ds_load_b128       v[56:59], v0 offset:192
ds_load_b128       v[48:51], v0 offset:208
ds_load_b128       v[60:63], v0 offset:224
ds_load_b128       v[64:67], v0 offset:240
s_waitcnt          lgkmcnt(0)
s_lshr_b32         s3, s6, 14
s_delay_alu        instid0(SALU_CYCLE_1) | instskip(NEXT) | instid1(SALU_CYCLE_1)
s_sub_i32          s8, s3, 1
s_and_b32          s3, s2, s8
s_delay_alu        instid0(SALU_CYCLE_1)
s_lshl_b32         s3, s3, 14
s_clause           0x3e
buffer_store_b32   v4, off, s[4:7], s3 glc
buffer_store_b32   v5, off, s[4:7], s3 offset:4 glc
buffer_store_b32   v6, off, s[4:7], s3 offset:8 glc
buffer_store_b32   v7, off, s[4:7], s3 offset:12 glc
buffer_store_b32   v16, off, s[4:7], s3 offset:16 glc
buffer_store_b32   v17, off, s[4:7], s3 offset:20 glc
buffer_store_b32   v18, off, s[4:7], s3 offset:24 glc
buffer_store_b32   v19, off, s[4:7], s3 offset:28 glc
buffer_store_b32   v8, off, s[4:7], s3 offset:32 glc
buffer_store_b32   v9, off, s[4:7], s3 offset:36 glc
buffer_store_b32   v10, off, s[4:7], s3 offset:40 glc
buffer_store_b32   v11, off, s[4:7], s3 offset:44 glc
buffer_store_b32   v20, off, s[4:7], s3 offset:48 glc
buffer_store_b32   v21, off, s[4:7], s3 offset:52 glc
buffer_store_b32   v22, off, s[4:7], s3 offset:56 glc
buffer_store_b32   v23, off, s[4:7], s3 offset:60 glc
buffer_store_b32   v24, off, s[4:7], s3 offset:64 glc
buffer_store_b32   v25, off, s[4:7], s3 offset:68 glc
buffer_store_b32   v26, off, s[4:7], s3 offset:72 glc
```

THE FUTURE OF THE GEOMETRY PIPELINE



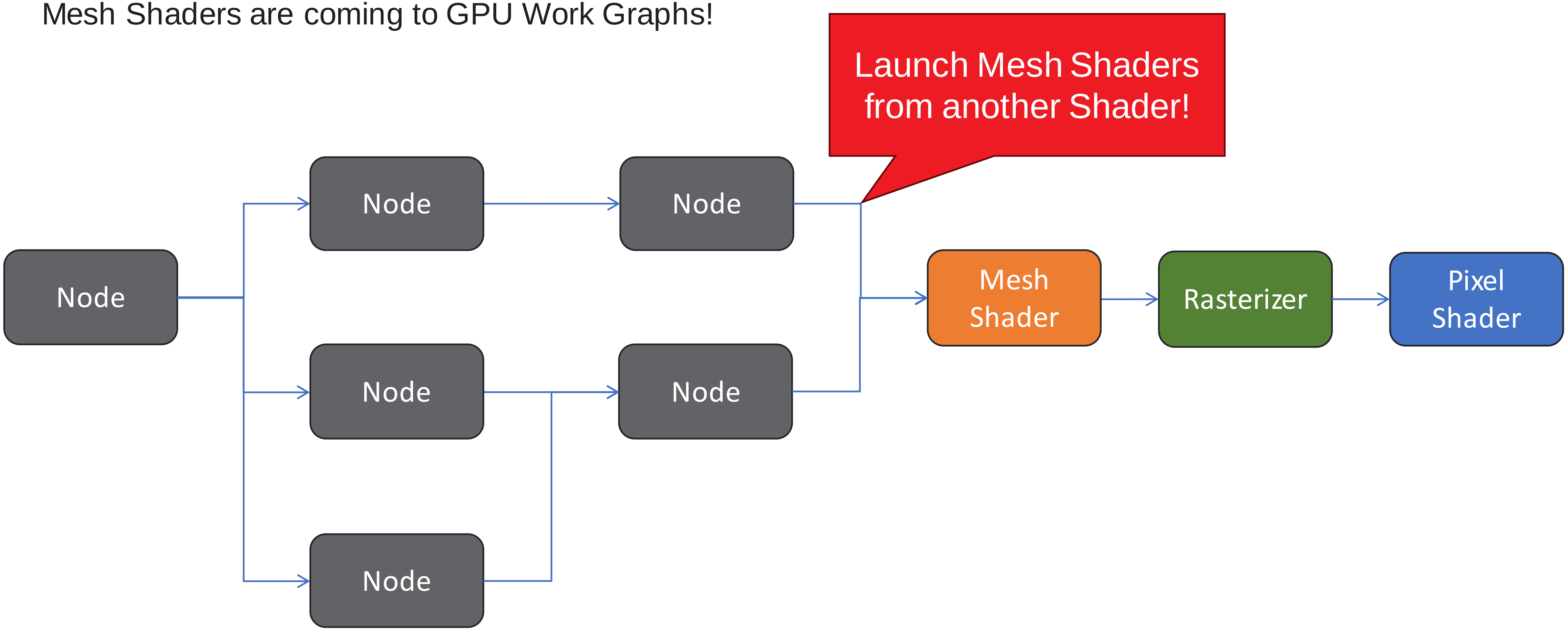
THE FUTURE OF THE GEOMETRY PIPELINE

Mesh Shaders are coming to GPU Work Graphs!



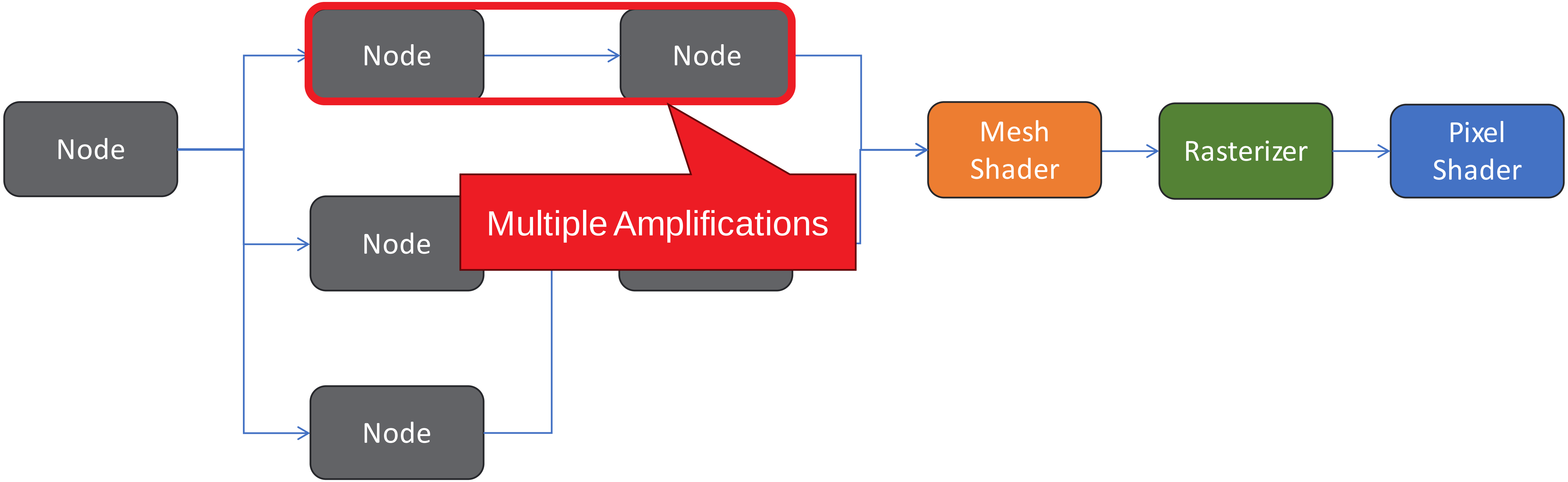
THE FUTURE OF THE GEOMETRY PIPELINE

Mesh Shaders are coming to GPU Work Graphs!



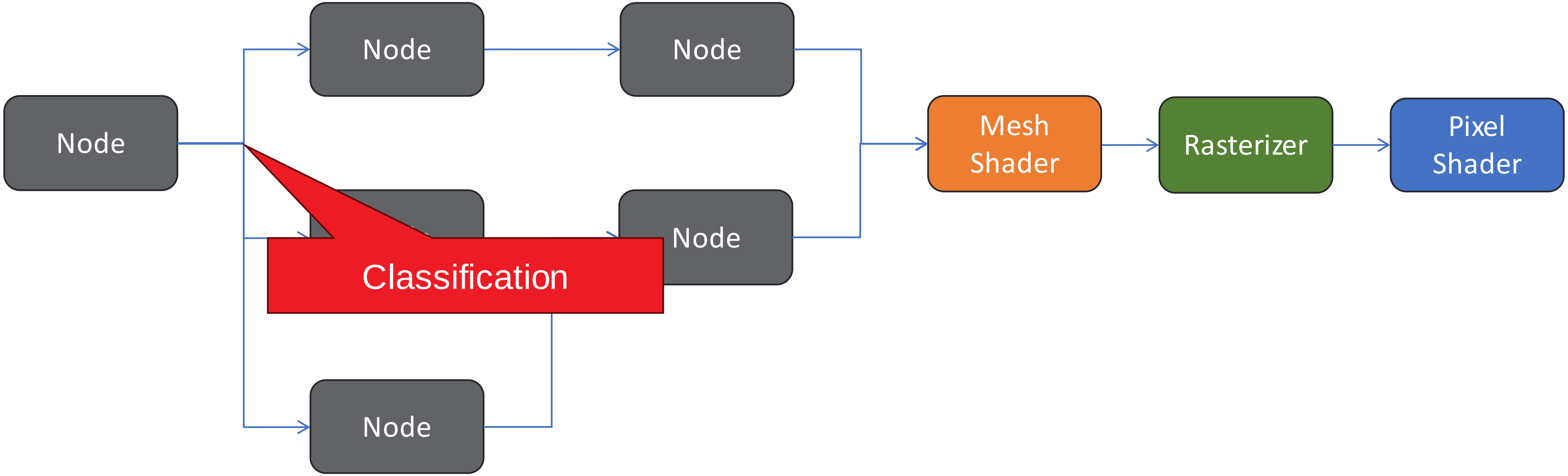
THE FUTURE OF THE GEOMETRY PIPELINE

Mesh Shaders are coming to GPU Work Graphs!



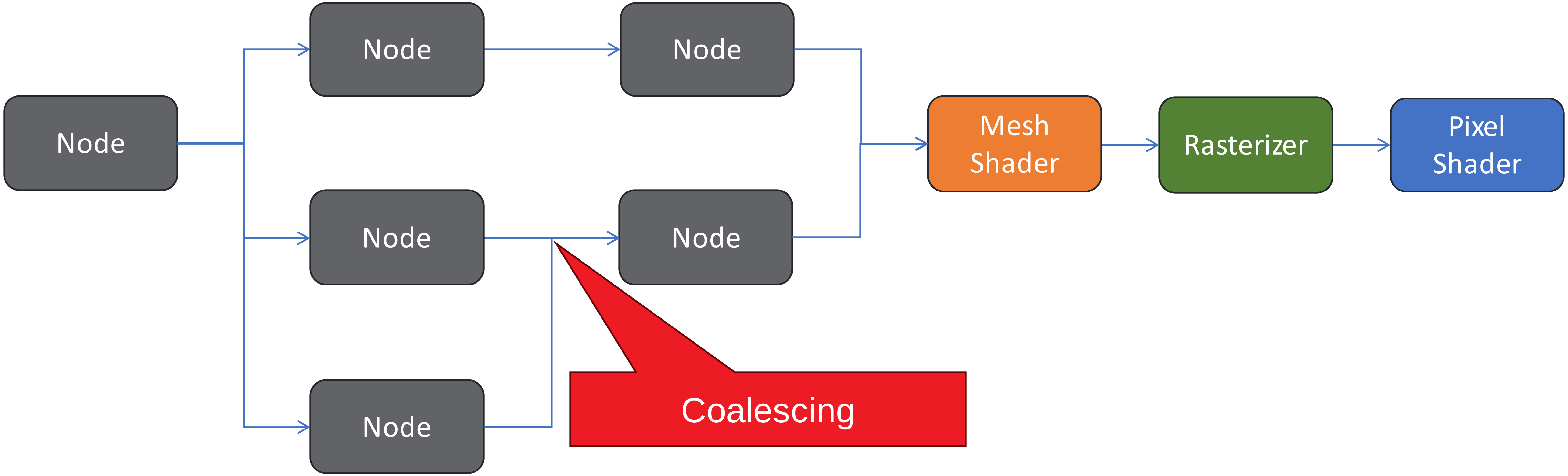
THE FUTURE OF THE GEOMETRY PIPELINE

Mesh Shaders are coming to GPU Work Graphs!



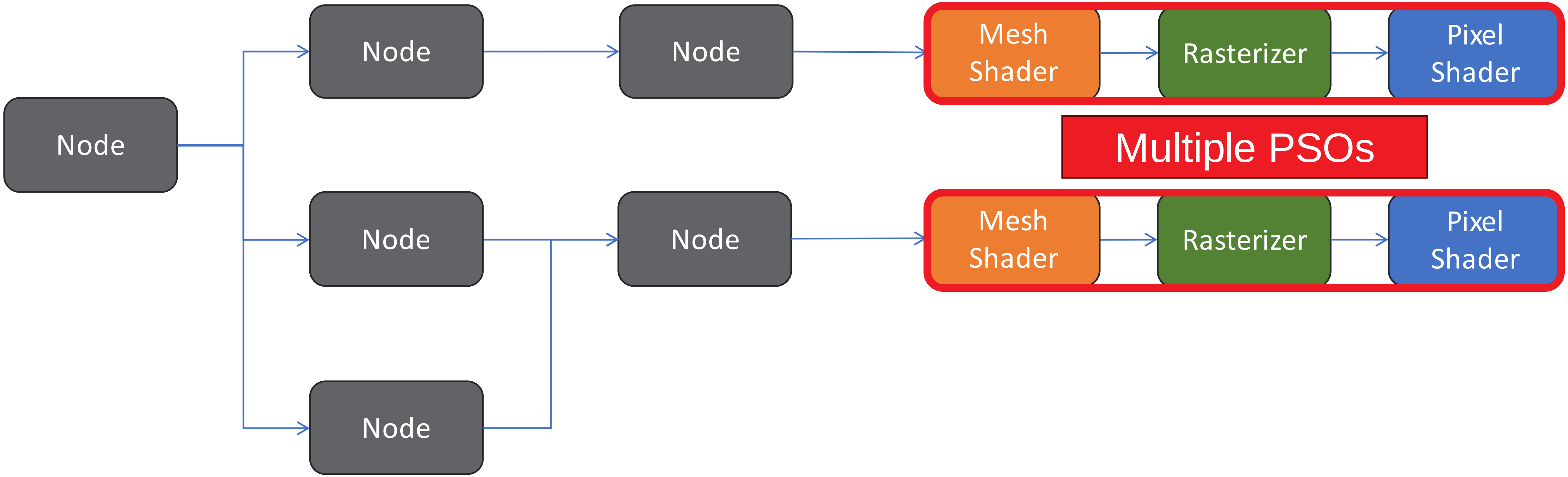
THE FUTURE OF THE GEOMETRY PIPELINE

Mesh Shaders are coming to GPU Work Graphs!



THE FUTURE OF THE GEOMETRY PIPELINE

Mesh Shaders are coming to GPU Work Graphs!



CONCLUSION

- Mesh shaders are a new way to process geometry
- Closer to the underlying hardware
- Allows for more control on how to use the hardware to process the geometry

- Can do things that were previously not possible
  - More opportunities for compression
- Can process any type of primitives, e.g. quads
- Can procedurally generate geometry in an efficient way

- For good performance, pay attention to
  - Meshlet generation
  - Vertex and primitive export
  - AS thread groups launching enough MS thread groups

SPECIAL THANKS

- Łukasz Izdebski
- Dawid Masiukiewicz
- Michael Grossfeld
- Todd Martin
- Dominik Baumeister
- Felix Kawala
- Pirmin Pfeifer
- Quirin Meyer
- Bastian Kuth
- Carsten Faber

DISCLAIMER

The information contained herein is for informational purposes only and is subject to change without notice. While every precaution has been taken in the preparation of this document, it may contain technical inaccuracies, omissions and typographical errors, and AMD is under no obligation to update or otherwise correct this information. Advanced Micro Devices, Inc. makes no representations or warranties with respect to the accuracy or completeness of the contents of this document, and assumes no liability of any kind, including the implied warranties of noninfringement, merchantability or fitness for particular purposes, with respect to the operation or use of AMD hardware, software or other products described herein. No license, including implied or arising by estoppel, to any intellectual property rights is granted by this document. Terms and limitations applicable to the purchase or use of AMD products are as set forth in a signed agreement between the parties or in AMD's Standard Terms and Conditions of Sale. GD-18

THIS INFORMATION IS PROVIDED 'AS IS." AMD MAKES NO REPRESENTATIONS OR WARRANTIES WITH RESPECT TO THE CONTENTS HEREOF AND ASSUMES NO RESPONSIBILITY FOR ANY INACCURACIES, ERRORS, OR OMISSIONS THAT MAY APPEAR IN THIS INFORMATION. AMD SPECIFICALLY DISCLAIMS ANY IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY, OR FITNESS FOR ANY PARTICULAR PURPOSE. IN NO EVENT WILL AMD BE LIABLE TO ANY PERSON FOR ANY RELIANCE, DIRECT, INDIRECT, SPECIAL, OR OTHER CONSEQUENTIAL DAMAGES ARISING FROM THE USE OF ANY INFORMATION CONTAINED HEREIN, EVEN IF AMD IS EXPRESSLY ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

Performance testing (slides 23, 30, and 31) done on following system: AMD Ryzen™ 9 7950X, AMD Radeon™ RX 7900 XTX, Asus ROG CROSSHAIR X670E HERO Motherboard, 128GB DDR5-6000 memory, 4TB M.2 NVME SSD, Windows® 11 Pro 22H2, AMD Software: Adrenalin Edition 23.40.14.01

Performance testing (slide 39) done on following system: AMD Ryzen™ 7 5800X, AMD Radeon™ RX 7900 XTX, AsusTeK TUF Gaming X570-Plus, 32GB DDR4-3600, Windows 10 Home 22H2, AMD Software: Adrenalin Edition 24.2.1

AMD GPUOpen

Visit our website

https://gpuopen.com

Follow us on X

https://twitter.com/GPUOpen

Follow us on Mastodon

https://mastodon.gamedev.place/@gpuopen

知乎

Follow us on Zhihu

https://www.zhihu.com/org/gpuopen-7

AMD
GPUOpen

AMD
together we advance_

AMD
EPYC

AMD
RYZEN

AMD
RADEON

AMD

together we advance_