MESH SHADERS IN TURING

Patrick Mours, 10/11/2018

AGENDA

Motivation

Research

Mesh Shaders

API Extensions

Examples

MOTIVATION

Detail Geometry

- Vegetation, undergrowth, greebles
- Fine geometric detail at massive scale
- Pre-computed topologies for LODs
- Efficient submission of small objects
- Flexible instancing

MOTIVATION

Auxiliary Meshes

- Proxy hull objects
- Iso-surface extraction
- Particles
- Text glyphs
- Lines/Stippling etc.
- Instancing of very basic shapes

MOTIVATION

CAD Models

- High geometric complexity
- Large assemblies can easily reach multiple 100 million triangles
- VR demands high framerates
- Cannot always rely on static solutions (animations, clipping etc.)

GENERAL DIRECTIONS

IMPROVE RAW RENDERING

Instancing, Multi-Draw-Indirect etc. already in use

Look into fixed function performance

IMPROVE GEOMETRIC PROGRAMMING

Geometry Shader (maybe not)

Tessellation Shader (great for certain patterns)

Compute Shader (generate rendered data)

HARDWARE

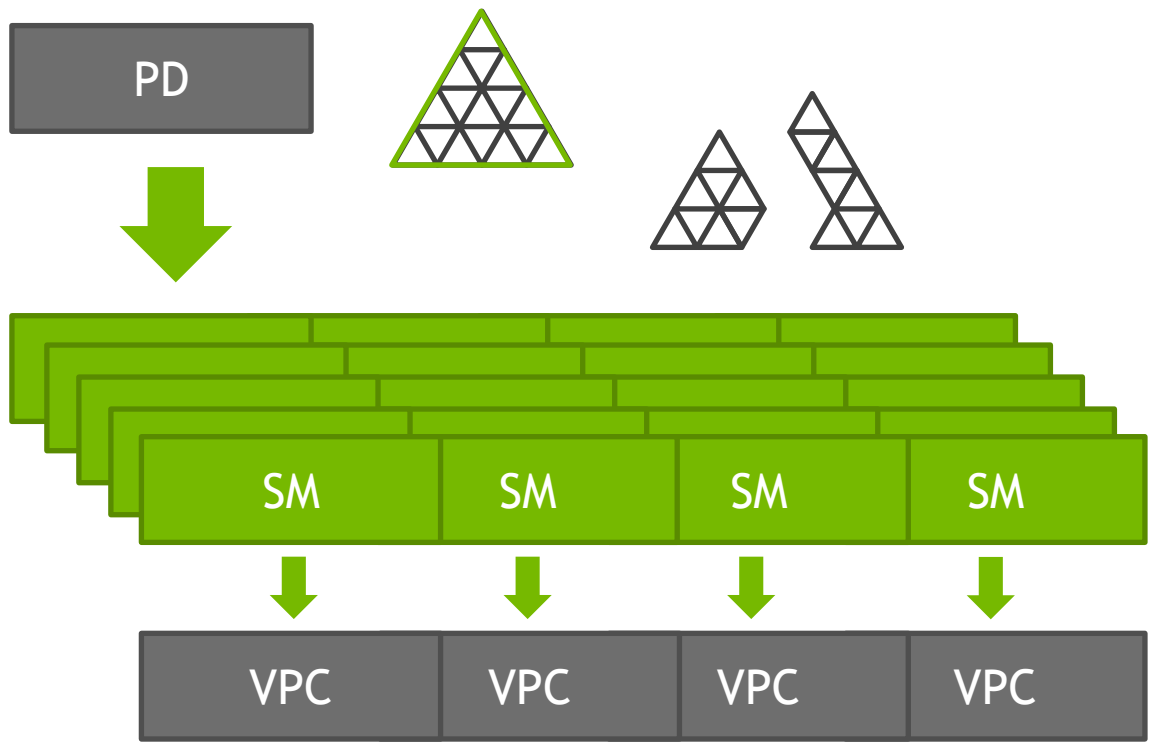
Vertex Pipeline

Fixed Function Blocks

Primitive Distributor (PD):
Creates vertex batches via serial indexbuffer processing

Viewport Culling (VPC):
Discards triangles (backface, frustum, samplegrid)

Let‘s use SM to do more work!
Easier to justify adding more of those ☺



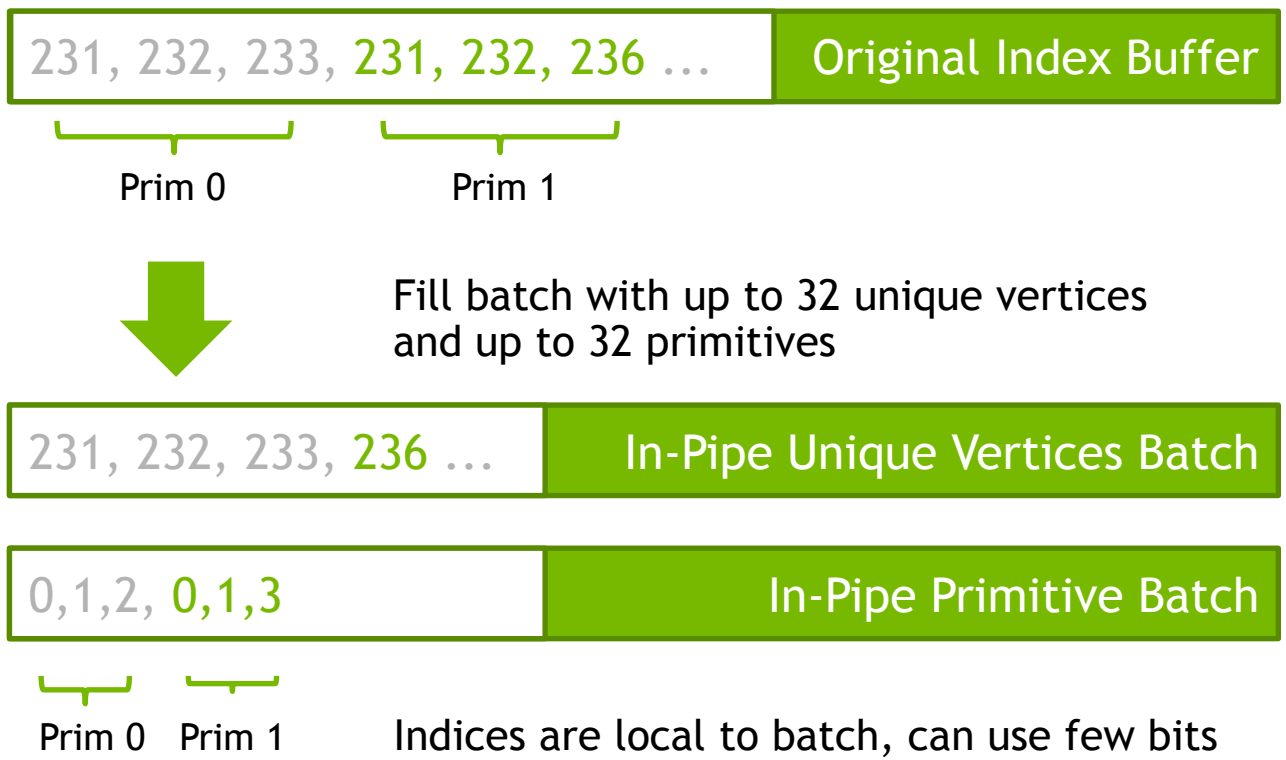
HARDWARE

Primitive Distributor

PD performs index de-duplication every frame

High bandwidth requirement

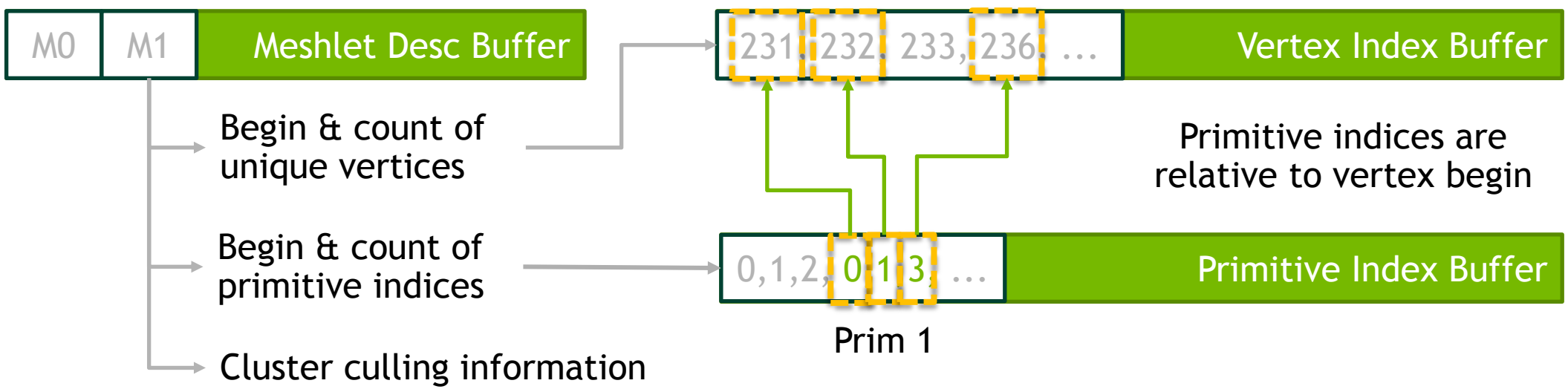
Wasteful for static topologies



MESHLETS

Pre-generate primitive batches for static topology and replace classic indexbuffer

Each meshlet can have up to a predefined maximum number of vertices and primitives



MESHLETS

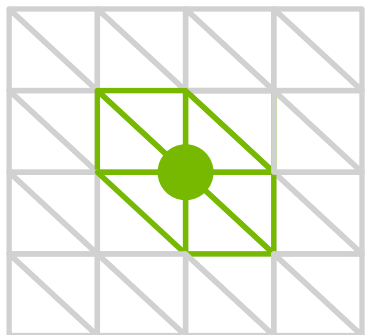
Allow more primitives than vertices

We can achieve higher vertex-reuse and save bandwidth

Twice as much triangles as vertices is ideal, we perceive average loads around 1.6x

Memory footprint of the three buffers is lower than classic indexbuffer!
(meshlet descriptor as single 128-bit value)

triangle count ~= vertex count
~ 3 triangles per vertex



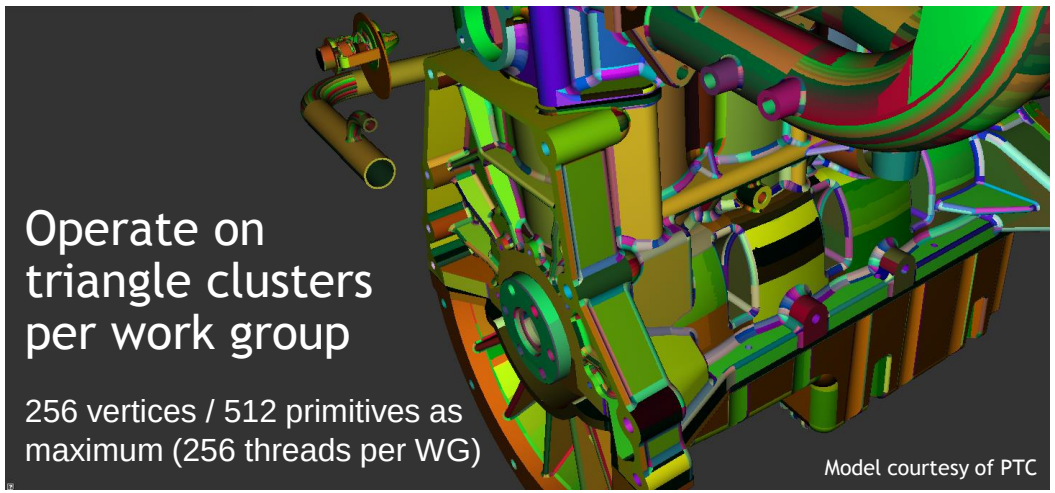
triangle count >> vertex count
~ 6 triangle per vertex

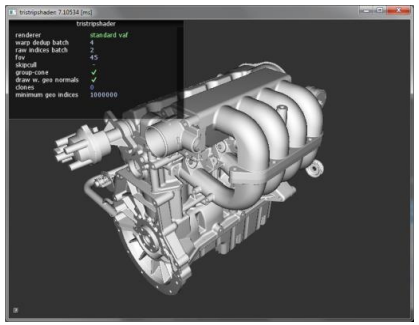
INDEX BUFFER BUILDING

Quadro Research

Use compute shader to build per object culled index buffers from meshlets

Similar to work by G.Wihidal [4], S.Aaltonen [5] and W.Engel [6].
They, however, mostly worked from regular indexbuffers





23 M triangles

Tech	Time [ms]
Standard	6.7
CS Meshlet cull / draw	2.6 2.2 / 0.4

Older HW generation

INDEX BUFFER BUILDING

Shader Phases

Compute provides flexible use of threads

Shared memory encodes the mesh‘s indices and vertices

```
// e.g. #Vertices = 256, #Primitives = 512
// WorkGroup = 256 threads

Load meshlet description
Cull entire cluster (optional early out)
Load block of primitive indices per thread
Load & transform 1 vertex per thread
Stored in shared memory

Barrier

Cull 2 triangles per thread &
Output surviving indices (local prefix sum etc.)
```

INDEX BUFFER BUILDING

Issues

PRO

Works universally, can beat fixed function on high rejection

Can save some bandwidth as unused vertex attributes are not fetched

CON

Compute launch overhead (need large jobs)

Storing of indexbuffer, refetching vertices (slower on low culling rate)

On-chip allocation for work group with fully culled clusters (registers etc.)

MESH SHADERS

MESH SHADERS

New programming model for geometry processing

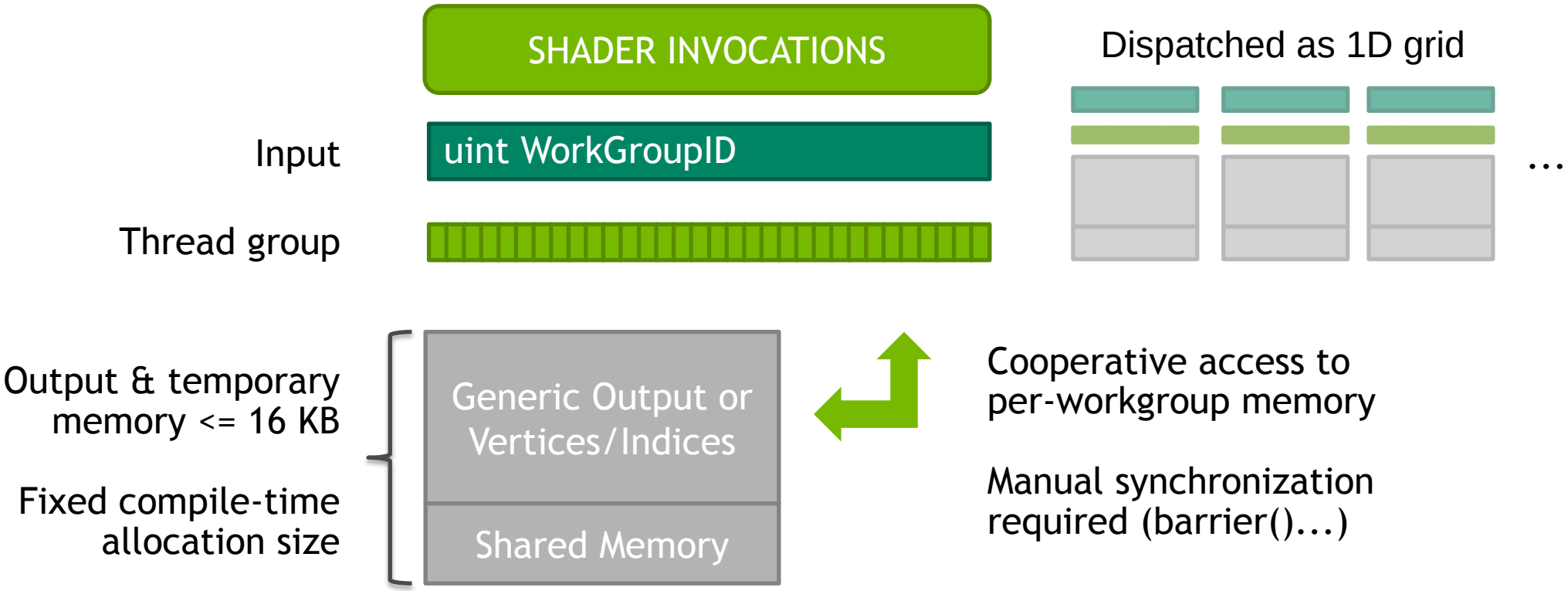
Evolution from singleton shaders to cooperative groups

- Pixel lighting ➔ Tile-based lighting via compute
- Vertex processing ➔ Meshlet processing

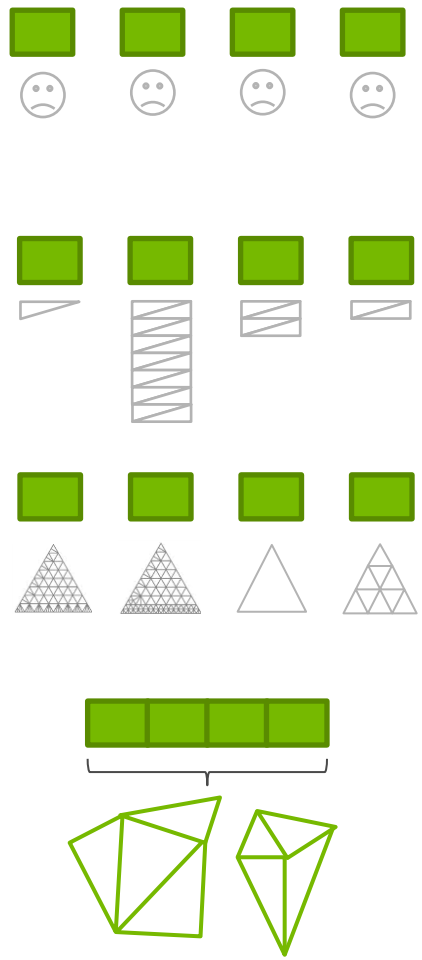
Essential components

- Compute-like execution model – data sharing and sync
- No fixed-function fetch for index processing or vertices
- Flexible work creation/tessellation with one level of expansion
- Standardized interface to screen-space pipeline

EXECUTION


SHADER INVOCATIONS
Dispatched as 1D grid
Input
uint WorkGroupID
Thread group
...
Output & temporary memory <= 16 KB
Fixed compile-time allocation size
Generic Output or Vertices/Indices
Shared Memory
Cooperative access to per-workgroup memory
Manual synchronization required (barrier()...)

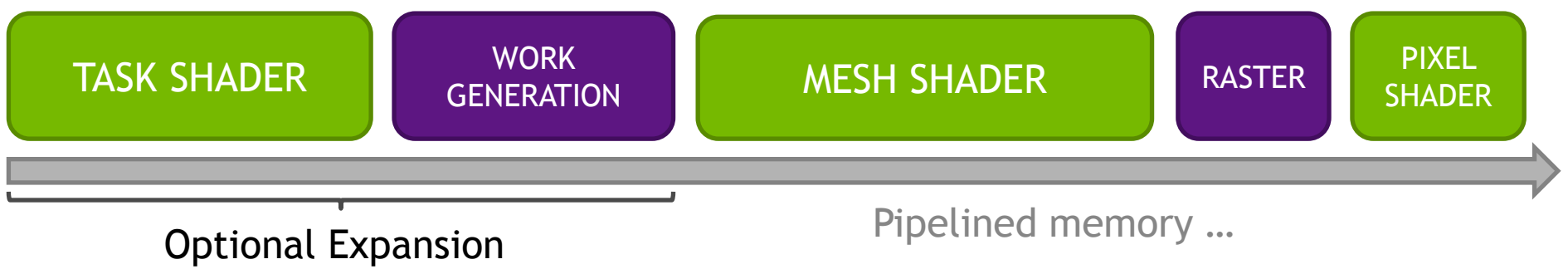
Shader		Thread Mapping	Topology
Vertex Shader	No access to connectivity	1 Vertex	No influence
Geometry Shader	Variable output doesn‘t fit HW well	1 Primitive / 1 Output strip	Triangle strips
Tessellation Shader	Fixed-function topology	1 Patch / 1 Evaluated vertex	Fast patterns
Mesh Shader	Compute shader features	Flexible	Flexible within work group allocation



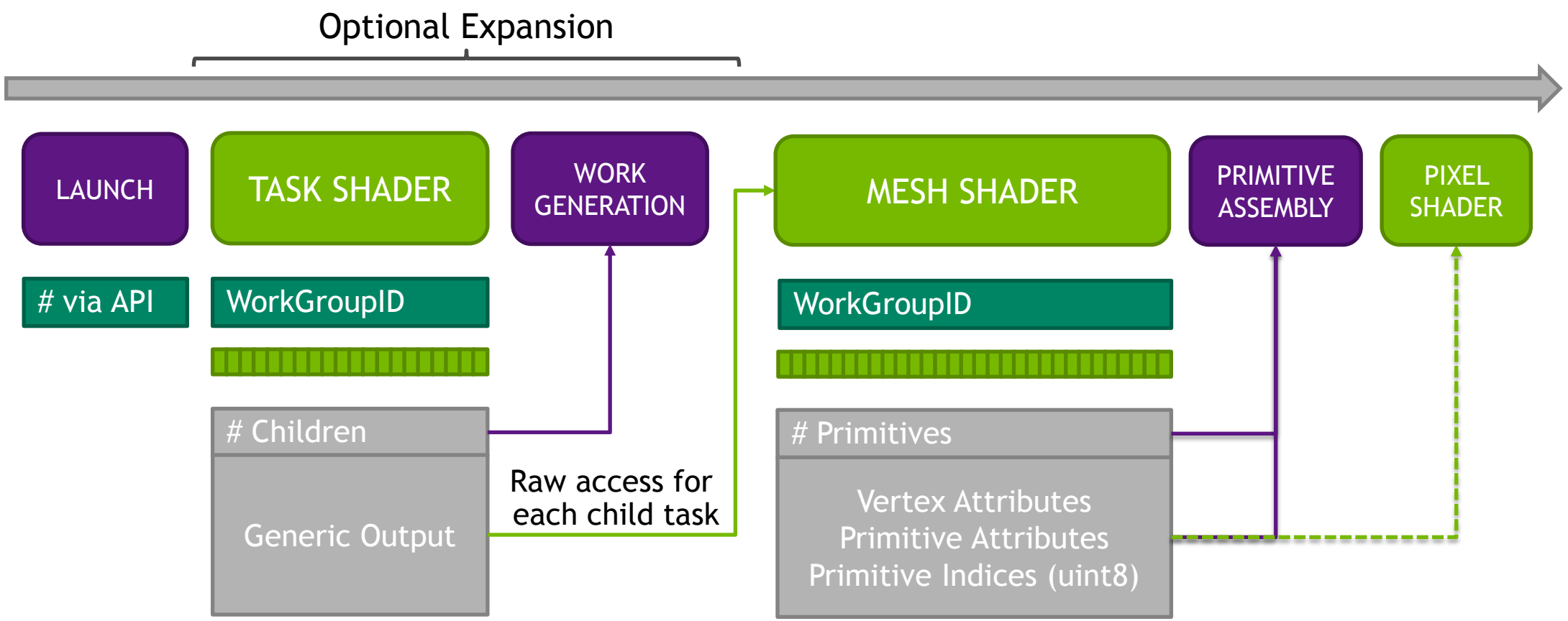
PIPELINE

TRADITIONAL VTG PIPELINE

TASK/MESH PIPELINE



PIPELINE


Optional Expansion
LAUNCH
TASK SHADER
WORK GENERATION
MESH SHADER
PRIMITIVE ASSEMBLY
PIXEL SHADER
via API
WorkGroupID
WorkGroupID
Children
Generic Output
Raw access for each child task
Primitives
Vertex Attributes
Primitive Attributes
Primitive Indices (uint8)

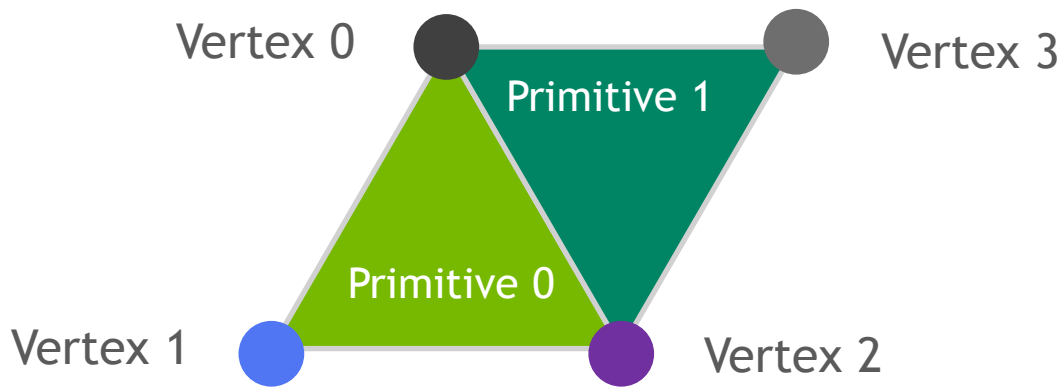
PIPELINE

Mesh Shader Stage

True per-primitive output attributes

Builtins like layer, viewport mask etc.

But also custom values and not just flat vertex attributes
(e.g. per-triangle normal, faceID, ...)



PIPELINE

Task Shader Stage

Task shader allows culling (subpixel, frustum, back-face, occlusion...)

Can also implement LOD picking to minimize mesh workgroups

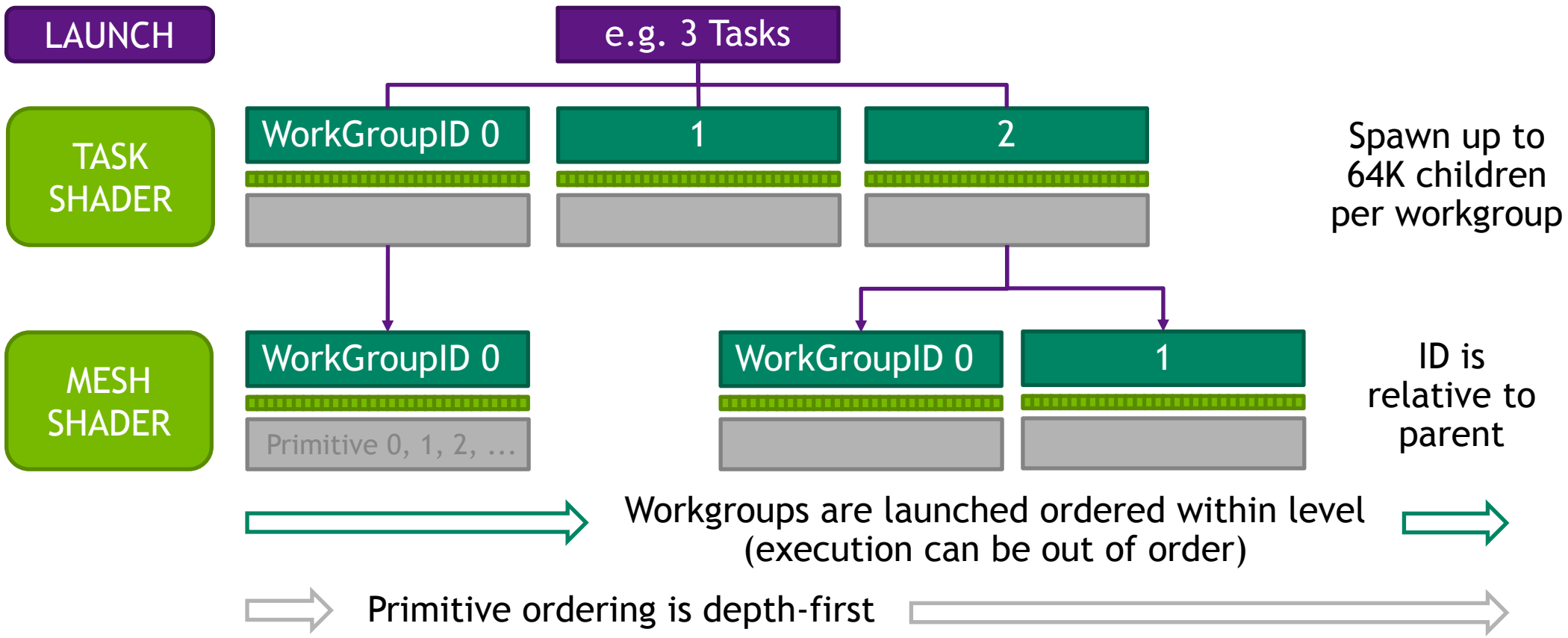
Not just about triangle models represented as meshlets

Use your own encodings for geometry, all data fetched by shader (compression etc.)

Provides more efficient procedural geometry creation (points, lines, triangles)

Can implement basic compute trees with disabled rasterizer

TREE EXPANSION



LIMITATIONS

Currently workgroup size = 1 warp = 32 threads

256 vertices maximum (recommended 64 vertices)

512 primitives maximum (allocation prefers 40, 84, 126, 254 or 510 triangles)

The smaller the outputs, the more warps can be in flight

Try to use small data types (barycentrics help with interpolation after unpack)

Avoid very inbalanced expansion trees

Microsoft® DirectX®

API

DX12 NVAPI

Pipelines with mesh shaders are created via NVAPI using a new mesh shader extension

```
void NvAPI_D3D12_CreateGraphicsPipelineState(
        ID3D12Device *pDevice,
        const D3D12_GRAPHICS_PIPELINE_STATE_DESC *pPSODesc,
        NvU32 numExtensions,
        const NVAPI_D3D12_PSO_EXTENSION_DESC **ppExtensions,
        ID3D12PipelineState **ppPSO);
```

API

GL & VK & SPIR-V EXTENSIONS

GLSL: *GL_NV_mesh_shader*, VK: *VK_NV_mesh_shader*

Introduces new graphics stages (**TASK**, **MESH**) that cannot be combined with VTG stages

New draw calls operate only with appropriate pipeline (similar calls in GL)

```
void vkCmdDrawMeshTasks(VkCommandBuffer buffer, uint32_t count, uint32_t first);
void vkCmdDrawMeshTasksIndirect(...);
void vkCmdDrawMeshTasksIndirectCount(...);
```

```
// Same as compute
layout(local_size_x = 32) in;
in uvec3 gl_WorkGroupID;
in uvec3 gl_LocalInvocationID;
...
shared MyStruct mySharedData;

// New for task shader
out uint gl_TaskCountNV;

// New for mesh shader
layout(max_vertices = 64) out;
layout(max_primitives = 126) out;
layout(triangles/lines/points) out;

out uint gl_PrimitivesCountNV;
out uint gl_PrimitiveIndicesNV[];
```

```
out gl_MeshPerVertex {
  vec4  gl_Position;
  float gl_PointSize;
  float gl_ClipDistance[];
  float gl_CullDistance[];
} gl_MeshVerticesNV[]; // [max_vertices]

perprimitiveNV out gl_MeshPerPrimitive {
  int gl_PrimitiveID;
  int gl_Layer;
  int gl_ViewportIndex
  int gl_ViewportMask;
} gl_MeshPrimitivesNV[]; // [max_primitives]

taskNV in/out MyCustomTaskData {
 ...
} myCustomData;
```

```
layout(local_size_x = 32) in;
layout(max_vertices = 32, max_primitives = 32, triangles) out;

out MyVertex { // Define custom per-vertex as usual (interfaces with fragment shader)
  vec3 normal;
} myOut[];

void main() {
  uint idx = gl_LocalInvocationID.x; uvec4 meshlet = meshInfos[gl_WorkGroupID.x];

  uint vertex = min(idx, meshlet.x - 1);
  myOut[idx].normal = texelFetch(texNormal, meshlet.y + vertex).xyz;
  gl_MeshVerticesNV[idx].gl_Position = texelFetch(texVbo, meshlet.y + vertex);

  uint primitive = min(idx, meshlet.z - 1);
  uint primitiveData = texelFetch(texTopology, meshlet.w + primitive);
  gl_PrimitiveIndicesNV[idx * 3 + 0] = (primitiveData <<  0) & 0xFF;
  gl_PrimitiveIndicesNV[idx * 3 + 1] = (primitiveData <<  8) & 0xFF;
  gl_PrimitiveIndicesNV[idx * 3 + 2] = (primitiveData << 16) & 0xFF;
  gl_PrimitiveCountNV = info.z; // Same for entire workgroup
}
```

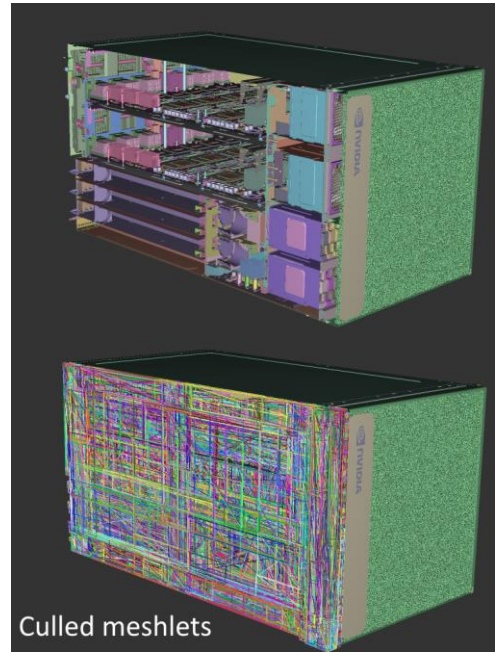
EXAMPLES

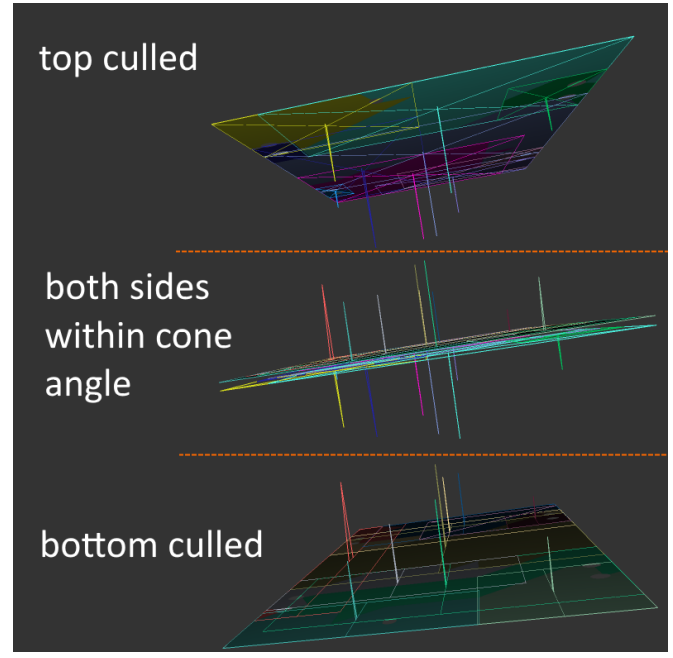
CAD SAMPLE

Open source sample

Task shader handles cluster culling:

- Outside frustum
- User clipping plane
- Back-face cluster
- Below custom pixel size





CAD SAMPLE

HW is still quite fast on triangle culling, focus on cluster culling

Separate positions from other interleaved attributes

Per-triangle culling may be benficial to skip loading fat vertices (high sub-pixel ratio)

```
taskNV out Task {
  uint baseID;
  uint8_t subIDs[GROUP_SIZE];
} OUT;

void main() {
  bool render = cull(gl_GlobalInvocationID.x);
  uvec4 renderBits = subgroupBallot(render);
  gl_TaskCountNV = subgroupBallotBitCount(renderBits);

  OUT.baseID = gl_WorkGroupID.x * GROUP_SIZE;
  uint idxOffset = subgroupBallotExclusiveBitCount(renderBits);
  OUT.subIDs[idxOffset] = uint8_t(gl_LocalInvocationID.x);
}
```

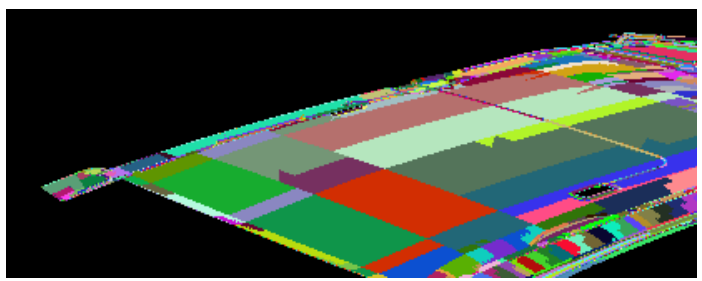
MESH OPTIMIZATION

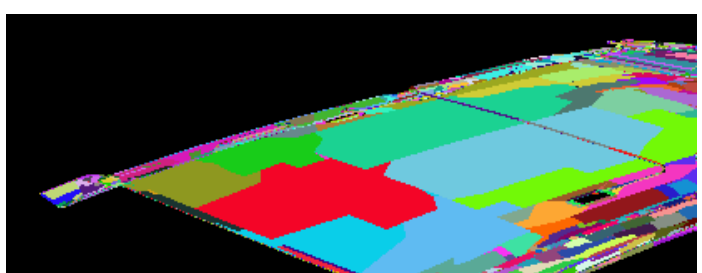
Maximize utilization of vertices and primitives

Use existing vertex cache optimizing algorithms to improve index buffers prior converting to meshlet

Various linear time algorithms exist, e.g. by T. Forsyth [7]
Caveat: may cause worse cluster culling (especially in CAD)

Meshlet compression (indices, vertices etc.)





TINY DRAW CALLS

Some scenes suffer from low-complexity drawcalls (< 512 triangles)

Task shaders can serve as faster alternative to Multi Draw Indirect (MDI)

- MDI or instanced drawing can still be bottlenecked by GPU
- Task shaders provide distributed draw call generation across chip
- Also more flexible than classic instancing (change LOD etc.)

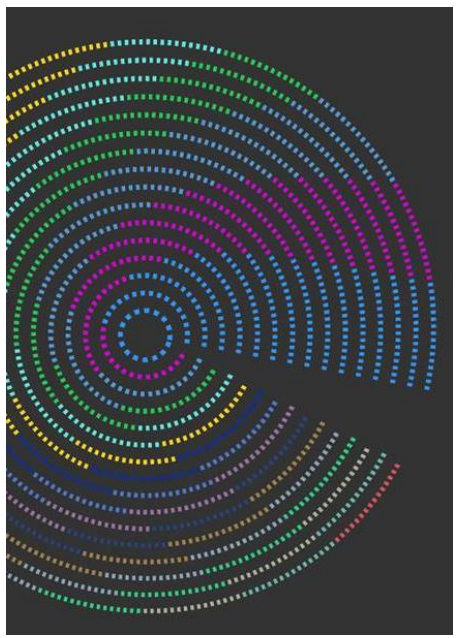
LINE STIPPLING

Task shader computes tessellation factor per curve/arc etc.

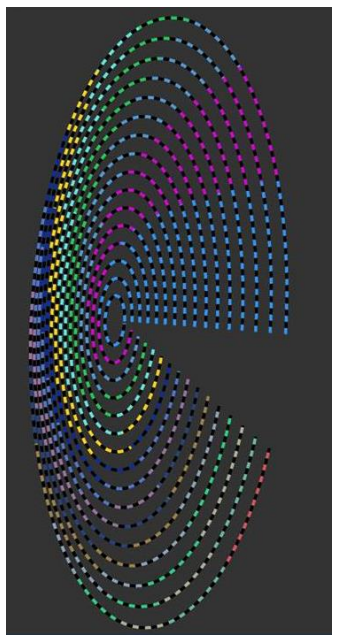
Mesh shader generates triangle strips

Use per-curve spin-lock for ordered append to compute running distance across child meshlets (workgroup launch ordering guarantees forward progress)

Custom line styles, caps, widths etc.
See GTC talk on shader-driven line rendering [8]



Geometry stipples



Texture space stipples

PROCEDURAL MESHES

Generate geometric shapes procedurally

Can use task shader to e.g. only create visibile meshes

```
void main() {
  const uint GRID = 5;
  uint vertIdx = gl_LocalInvocationID.x;
  vec4 gridPos = vec4(vertIdx % GRID, vertIdx / GRID, 0, 1);
  gl_MeshVerticesNV[vertIdx].gl_Position = mvp * gridPos;

  uint primIdx = gl_LocalInvocationID.x * 6;
  uint primOffset = primIdx / (GRID – 1);
  gl_PrimitiveIndicesNV[primIdx + 0] = primOffset;
  gl_PrimitiveIndicesNV[primIdx + 1] = primOffset + GRID;
  gl_PrimitiveIndicesNV[primIdx + 2] = primOffset + GRID + 1;
  gl_PrimitiveIndicesNV[primIdx + 3] = primOffset;
  gl_PrimitiveIndicesNV[primIdx + 4] = primOffset + GRID + 1;
  gl_PrimitiveIndicesNV[primIdx + 5] = primOffset + 1;

  gl_PrimitiveCountNV = (GRID - 1) * (GRID - 1) * 2;
}
```

CONCLUSION

Mesh & task shaders are a flexible new tool to create geometry on the chip

Built on top of proven technology of the hardware‘s tessellation pipeline

Supplementary to existing pipeline strengths

Better scalability for low-complexity geometry or high-complexity fixed topologies

THANK YOU

```
contributors[] = {
 // research & GLSL
 Christoph Kubisch,
 // guidance
 Pierre Boudier,
 // programming model
 Yury Uralsky,
 Henry Moreton,
 // hardware design
 Ziyad Hakura,
 // hardware early tests
 Samba Wen,
 // VK/GL/GLSL/SPIR-V driver
 Pat Brown,
 Tyson Smith,
 Daniel Koch,
 Jeff Bolz,
 // SPIR-V,GLSL compiler
 Sahil Parmar,
 // Gaming & DX NVAPI
 Manuel Kraemer,
 Iain Cantlay,
 Gabriel Liberty,
 Rahul Sathe,
 ... many more
};
```

DevTech Contacts
ckubisch@nvidia.com & pmours@nvidia.com (professional use, GL/VK)
mkraemer@nvidia.com (gaming use, DX/NVAPI)

[1] www.facebook.com/artbyrens

[2] https://www.flickr.com/photos/14136614@N03/6209344182

[3] k-DOPs as Tighter Bounding Volumes for Better Occlusion Performance – Bartz, Klosowski & Staneker
https://pdfs.semanticscholar.org/bf4e/7c405d0f2a259f78e91ce1eb68a5d794c99b.pdf

[4] GDC 2015 – Optimizing Graphics Pipeline with Compute – Graham Wihlidal
https://frostbite-wp-prd.s3.amazonaws.com/wp-content/uploads/2016/03/29204330/GDC_2016_Compute.pdf

[5] SIGGRAPH 2015 - GPU-Driven Rendering Pipelines – Ulrich Haar & Sebastian Aaltonen
http://advances.realtimerendering.com/s2015/aaltonenhaar_siggraph2015_combined_final_footer_220dpi.pdf

[6] GDC EU 2016 – The filtered and culled Visibility Buffer – Wolfgang Engel
http://www.conffx.com/Visibility_Buffer_GDCE.pdf

[7] https://tomforsyth1000.github.io/papers/fast_vert_cache_opt.html

[8] GTC 2016 – OpenGL Blueprint Rendering – Christoph Kubisch
http://on-demand.gputechconf.com/gtc/2016/presentation/s6143-christoph-kubisch-blueprint-rendering.pdf

[9] https://www.artstation.com/artwork/rWEba


NVIDIA