Turing Mesh Shaders

Viktor Zoutman (viktor@vzout.com)

30 January 2020

Abstract

In 2018 NVIDIA Released their GPU architecture, called Turing[1]. Its main feature is ray-tracing[2] but Turing also introduces several other developmenhttps://www.wikihow.com/Write-an-Abstractts[3] that look very intereresting on their own. One of these developements is the concept of *mesh shaders*. But what are mesh shaders and how can we improve use them to improve your renderear?

```
\renewenvironment{Shaded}
{\begin{snugshade}\footnotesize}
{\end{snugshade}}
```

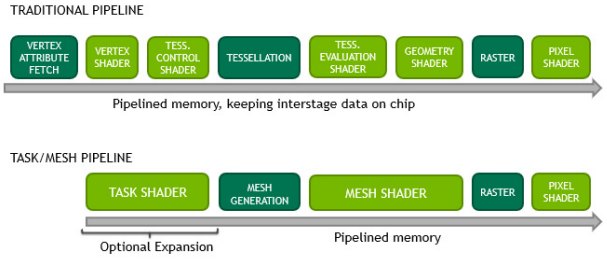
NOTE: ***Rewrite the abstract when the paper is finished.***

NOTE: ***Figure out how to change 'lst [N]' into 'listing [N]'.***

NOTE: ***Reference api functions/intrinsics I mention within the text like 'SetMeshOutputCounts'***

1 Introduction

Over time the graphics pipeline has gotten more and more complicated. While some parts of the currect graphics pipeline are flexible (geometry shaders, tesselation) they are not performant and where the graphics pipeline is performant it is often not flexible (instancing).



Mesh shaders aim to simplyify the graphics pipeline by removing the input assembler, replacing the tesslator with a mesh generator, replacing the vertex shader and tesselation control shader with a (optional) *task shader* (Called *amplification shader* in DirectX 12). and the tesselation evaluation shader and geometry shader with a mesh shader. This simplification has the effect of introducing higher scalability and bandwidth-reduction.

NOTE: ***Explain why mesh shading is more scalable and can reduces bandwidth.***

This allows graphics programmers to satisfty the need for the high poly count and high number of objects in modern video games and graphics software like CAD.

NOTE: ***Write about subgroups and wave intrinsics in a background information or when discussing optimization of mesh shaders.***

1.1 The Mesh Shader

The mesh shader (and task shader) are basically compute shaders. A mesh shader begins its work by dispatching a set of threadgroups, each of which process a subset of the larger mesh. Similar to compute each thread has access to groupshared memory. The output vertices and primitives however do not have to corrolate to a specific thread in the group. As long as all vertices and primitives used in the threadgroup are processed, resources can be allocated in whatever way is most efficient. The user is also capable of specifying per-vertex and per-primitive attributes, which allows for faster and space efficient rendering.

There are currently 2 implementattions of mesh shaders. One in the DirectX 12 API and one as a Vulkan Extension. There are a couple of differences between these. Vulkan allows only 1 dimensional work groups. DirectX however allows for 3 dimensional work groups. DirectX also has the ability to dynamically specify the number of output vertices dynamically unlike Vulkan which only allows you to specify that value statically.

The aforementioned submeshes processed by thread groups are called *meshlets*. The idea is that the programmer algorithmicly splits the mesh in x amount of meshlets with a vertex count of 32 to around 200 vertices, depending on the number of attributes. It is most efficient to generate meshlets with as many as possible vertices that allow vertex re-use. These meshlets should be pre-computed. This is a big benefit over the old Input Assambler which has to identify vertex re-use dynamically. In sec. 2 I'll go over how to efficiently pre-compute meshlets.

1.2 The Task Shader

An optional expansion in the mesh shader pipeline is the *task shader* (*amplification shader*). While the mesh shader is a flexible tool it does not allow for tessellation or efficient culling of entire meshlets.

The most basic use of a task shader is basically executing mesh shaders. We can determine for example whether a meshlet is visible and conditionally execute the mesh

shader that is supposed to process the meshlet. Task shaders are capable of sharing data with its child mesh shaders and also the childs have access to the parents group shared memory. This allows you to add more triangles to the meshlet for displacement mapping for example since you can now exceed the limited vertex count per mesh shader by executing 2 mesh shaders instead.

1.3 Executing Mesh & Task Shaders

Both Vulkan and DirectX 12 allow you to either execute mesh shaders directly or using execute indirect. Vulkan has the function `CmdDrawMeshTasksNV` with the parameters `uint32_t taskCount` and `uint32_t firstTask`. firstTask allows you to specify what mesh shader to execute first. Lets say you have a task shader and a mesh shader but they are placed in the pipeline in the order of mesh shader, task shader but you still want to execute the task shader first. In this case you can set `firstTask` to 1 and it will execute the task shader first followed my the mesh shader. The other parameter - `taskCount` - is a bit misleading. It is actually the amount of work groups in the `x` dimension you want to execute on the first task shader.

DirectX 12 is a bit more straight forward with its `DispatchMesh` function. It has the parameters `ThreadGroupCountX`, `ThreadGroupCountY` and `ThreadGroupCountZ` parameters. Each thread group size must be less than 64k and the product of $ThreadGroupCountX \times ThreadGroupCountY \times ThreadGroupCountZ$ must not exeed 2^{22}

In Vulkan to calculate the `numTasks` parameter I use the following formula: $numTasks = (N+S-1)/S$ where N is the number of meshlets multiplied by the number of instances and S is the size of the work group. DirectX is a bit more complicated since you could optimize the mesh shader in different ways using the 2 extra dimensions. For example every instance of a meshlet can be a index in the `y` dimension.

NOTE: ***Possibily explain the differences between direct and indirect execution.***

You don't need to bind a vertex buffer the traditional way anymore. Instead you are required to create a descriptor to your buffers and use that to read from the buffer directly in the mesh shader. You could just bind the vertex buffer and index buffer directly without modifying the contents of it but this is not the most efficient approach. For these optimizations see sec. 2.

2 Generating Meshlets

First of all we want to have vertices and indices we can process in the mesh shader. Remember we want to optimize the re-use of vertices so every meshlet should have the highest number of re-use possible. We can build the vertex buffer as followed:

1. Take the first triangle that hasn't already been added to a meshlet.
2. Loop over all triangles left in the mesh.
   - Find as many duplicate vertices as possible and add their parent triangle to the meshlet (without exeeding the maximum amount of vertices).
   - If there are no duplicates left add any triangle that hasn't been added to a meshlet until the maximum amount of vertices is reached.

Keep in mind that theoratically it is possible to have a vertex re-used more often than the maximum vertex count of a meshlet. Also note that you can't share a single triangle over 2 meshlets.

While you assigning vertices to meshlets you can also create the index buffer of the meshlets. We can store the indices like normal and index from the start to the end of the vertex buffer. Or you can flatten the index buffer and in the mesh shader append a "vertex start" variable to the index. This allows us to have a 32 bit index buffer where we store 4 indices in 1 value and write with one operation 4 indices to the output of the mesh shader (`writePackedPrimitiveIndices4x8NV`).

3 Rendering Meshlets

> In this section I'll focus on Vulkan and mostly ignore DirectX12. Vulkan's version of wave intrinsics is called *subgroups*. You can read more about them here 4 and see how they compare to DirectX12's wave intrinsics 5.

So first things first. We need to decide on the work group size. I found that using all the threads available in the warp (wavefront) was most efficient (32 in the case of my RTX 2060).

To generate our primitives in the mesh shader we require some data. For now we only need the number of vertices in the meshlet, the number of primitives in the meshlet, the offset of the meshlet in the vertex buffer and the offset of the meshlet in the index buffer. In my testing I store the counts in 8 bits and the offsets in 20 bits. This data is our meshlet descriptor. We want to access these descriptors on the GPU so we need to store them in a buffer. On the GPU we can index into the buffer using the following formula: $i = base\_id + lane\_id$ where $base\_id = work\_group\_id_1 \times work\_group\_id_2 \times work\_group\_id_3$ and $lane\_id = local\_invocation\_id_1 \times local\_invocation\_id_2 \times local\_invocation\_id_3$

In the mesh shader you would want to have two loops. One parsing the indices and one parsing the vertices. You could do this in the same loop. But since we likely have duplicate vertices we don't want to parse vertices twice or introduce branching. We want to split the work over the work group. We can calculate the number of itterations for the loops respectively with the following two formulas: $prim\_count/group\_size$ and $vert\_count/group\_size$. To allow the shader compile

to compute this during compilation we can use the maximum primitive and vertex count instead of the actual numbers. This has the additional benefit of reduced branching.

Parsing the vertices should be relatively straight forward. Output the vertex position (multipled by the MVP if required) and output the other attributes of the vertex (normals, tangents, etc).

Parsing indices is even simpiler but instead of just outputting the indices directly we can optimize it by writing 4 or 8 indices in the same loop iteration. If you want to make use of this optimization the formula used to calculate the number of loop iterations becomes $prim\_count/group\_size \times indices\_per\_iteration$. You can write 4 indices at the same time by using `writePackedPrimitiveIndices4x8NV` in GLSL. HLSL however does not have such function.

NOTE: ***difference between 4 vs 8 indices per iteration***

When parsing meshlets it is important to understand that it doesn't matter what vertices and indices you are parsing per thread. If you parse x index you don't require its corresponding vertex in the same thread. This allows us to parse vertices and indices in any order.

In the vertex and index parsing loops which iterate `i` from 0 to `num iterations` (see blahblah) we can calculate the index using the following formula: $local\_invocation\_id\_1 + i * group\_size$. Meaning we loop over a subset of indices and vertices per lane so we don't parse indices or vertices multiple times within the same thread group.

4 Instancing

Instancing is a bit more difficult compared to the old vertex pipeline. I have 2 solutions for this problem. One is uses Execute Indirect for instancing while the other does the instancing with task shaders. Both techniques still require you to bind a big buffer with all the per-instance data just like the old days.

4.1 Instancing Using Execute Indirect

This is a rather simple approach. The mesh shader API provides us with `blahblah` and works similar to normal indirect execution. And instead of calculating the instance id ourselves we can use the `ARB_shader_draw_parameters` extension which provides us with `gl_DrawIDARB` which we can functionally use as the instance id.

4.2 Instancing Using Only Mesh

We need to calculate `gl_InstanceID` ourselves and instead of increasing the `num instances` parameter of the old draw command we need to multiple the number of meshlets per mesh by the number of instances.

First off lets compute the instance id. For this we need to know the number meshlets of the mesh which we need for the following formula: $inst\_id = (base\_id + lane\_id)/num\_meshlets$. We will need to adjust the meshlet id calculation as well: $meshlet\_id = base\_id + lane\_id \pmod{num\_meshlets}$

4.3 Benchmarks

5 Culling

5.1 Benchmarks

6 Tessellation

6.1 Benchmarks

7 Raytracing

I had hoped mesh shading would go hand in hand with ray tracing. But this doesn't seem to be the case. Of course mesh shading can still be used to accelerate hybrid raytracing but you can't reuse the meshlet buffers and descriptions in a valuable way for the acceleration structures or obtaining attributes (uv, normals and etc) in the raytracing shaders.

8 Concolusion

9 Further Work

There are undoubtedly many more optimization and techniques yet to be discovered. Some subjects that require research but not limited to are:

- Using wave intrinsics to share vertex and index data between instances.
- Benchmark how the `SetMeshOutputCounts`'s num vertices affects performance compared to Vulkan's approach.
- Dynamic Level of Detail approaches.
- Procedual geometry.
- Execute-Indirect and mesh shaders.
- My current approach to displacement mapping is very basic. This could undoubtedly be improved drastically.

References

1. NVIDIA Corporation. Turing architecture. **https://www.nvidia.com/en-us/design-visualization/technologies/turing-architecture/**, (2018).

2. NVIDIA Corporation. Raytracing. **https://developer.nvidia.com/rtx/raytracing**, (2018).

3. NVIDIA Corporation. Turing architecture in-depth. **https://devblogs.nvidia.com/nvidia-turing-architecture-in-depth/**, (2018).

4. Daniel Koch, NVIDIA. Vulkan subgroup explained. **https://www.khronos.org/assets/uploads/developers/library/2018-vulkan-devday/06-subgroups.pdf**, (2018).

5. Microsoft Corporation. Wave intrinsics. **https://github.com/Microsoft/DirectXShaderCompiler/wiki/Wave-Intrinsics**, (2018).