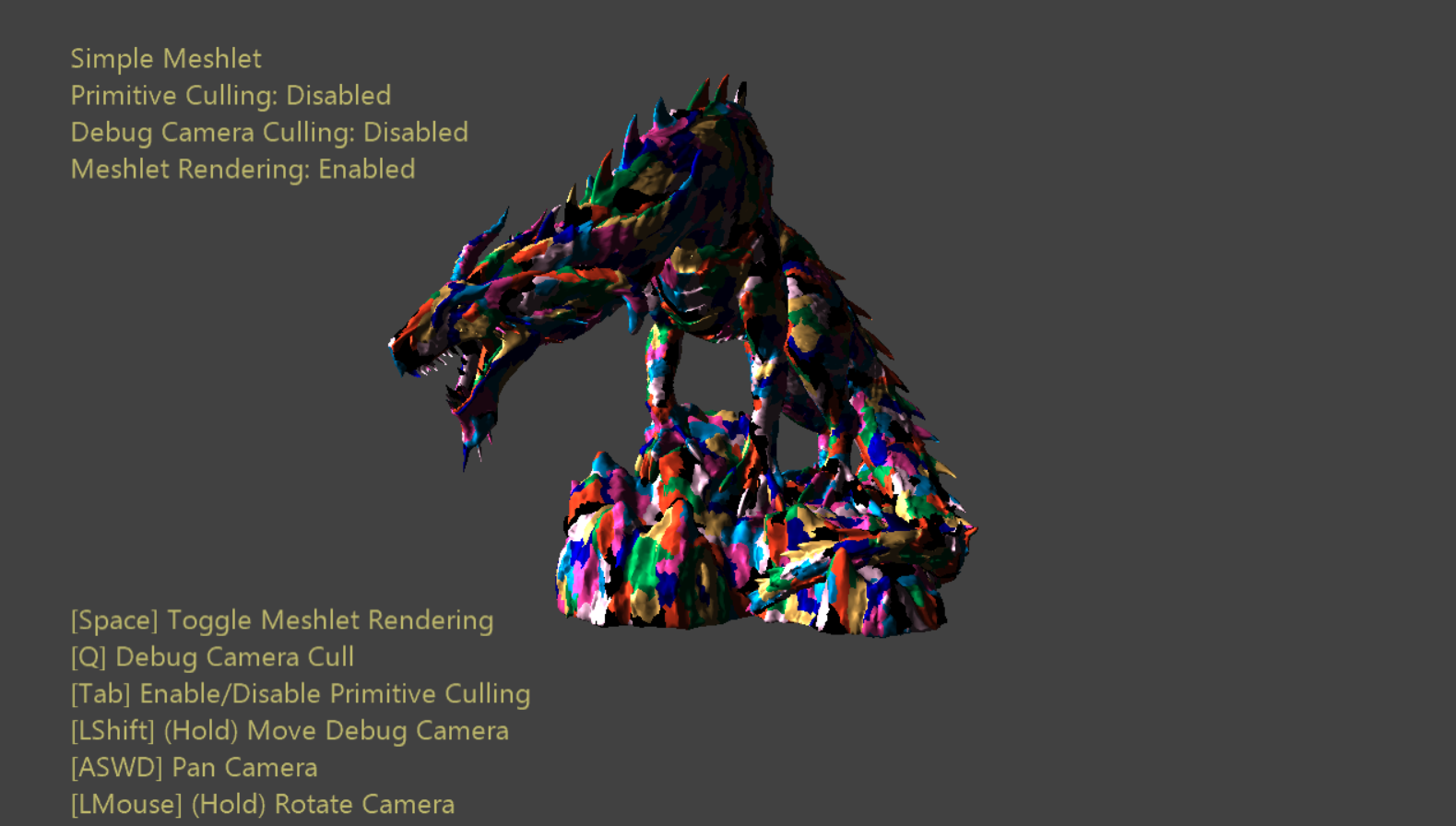


Simple Meshlet Sample

*This sample is compatible with the Microsoft Game Development Kit (June 2020) and Windows 10 (Version 2004) May 2020 Update*

# Description

This sample introduces the meshlet data structure and provides an example of rendering using meshlets. It also demonstrates how to do primitive culling inside of mesh shaders.



# Building the sample

If using Project Scarlett, set the active solution platform to Gaming.Xbox.Scarlett.x64.

If using PC with appropriate hardware and Windows 10 release, set the active solution platform to Gaming.Desktop.x64.

This sample does not support Xbox One.

*For more information, see* Running samples*, in the GDK documentation.*

# Using the sample

Beyond camera controls there a few options that have been exposed with which to be toyed.

A visual representation of the underlying meshlet structure can be toggled with the click of a button – each colored patch represents a meshlet of maximum size 128.

Primitive culling can also be toggled at will. A ‘debug’ camera has been placed in the scene which may be optionally used as the view against which primitives are culled. This allows the user to visualize the culled primitives. This camera’s position & orientation can be manipulated by holding down a button and manipulating the camera controls.

# Controls

|  |  |  |
| --- | --- | --- |
| Action | Gamepad | Keyboard |
| Rotate/translate camera along view vector | Left Thumbstick | Mouse wheel |
| Orbit camera | Right Thumbstick | Hold LMB + Mouse |
| Pan Camera | Directional Pad | WASD or Arrow Keys |
| Reset camera | Right Thumbstick (Push) | - |
| Toggle Meshlet Visualization | X | Spacebar |
| Toggle Primitive Culling | A | Tab |
| Toggle Debug Camera Cull | B | Q |
| Debug Camera Control (Hold) | Right Shoulder | Left-Shift |
| Cycle Mesh LODs | Left/Right Triggers | Plus/Minus Keys |
| Exit | View Button | Escape |

# Implementation notes

**Meshlets** are exactly what they sound like – fixed-sized primitive chunks of a larger mesh. A maximum size is chosen for the meshlet structure, then primitives & vertices are packed into meshlets until the entire mesh has been processed. In this way a mesh 🡪 an array of meshlets.

Note that the actual vertex data is unchanged by this process, but the index buffer is replaced by three new buffers – a *meshlet list*, a *unique vertex index list*, and a *primitive list*. The elements of the *meshlet list* are simple offsets & counts into the other two structures – this defines which vertices & primitives are in each meshlet. The *unique vertex index list* contains chunks of de-duplicated vertex indices for each meshlet – these are used to directly index into the vertex buffer. The *primitive list* defines chunks of primitives for each meshlet. The items of this list are indices into the unique vertex index list. Each primitive index is local to the meshlet’s unique vertex index sub-range which reduces its range to only 8-bits.

This structure maps very well to the fixed-sized threadgroups of mesh shaders – each meshlet can be mapped to a single threadgroup. Each meshlet has a fixed maximum size, so this correlates nicely with the work for which each thread will be responsible. This is the basis of the shader BasicMeshletMS.hlsl, and is very straightforward implementation-wise.

**Primitive culling** is the process of determining viewport relevance per-primitive against several culling tests. Since mesh shaders dynamically specify their output counts, discarding primitives is done by simply omitting them from submission. The basic workflow for mesh shader based primitive culling is the following steps:

1. Transform meshlet vertices into cull space (generally view, homogeneous, or NDC.)
2. Build primitive from transformed vertices & do cull tests
3. Mark surviving primitives 🡪 mark surviving vertices
4. Determine final output indices of vertices & primitives via **compaction**
5. Remap primitive indices to remapped vertex indices
6. Export as usual

**Compaction** is an algorithm that produces a sparse list which indexes into a dense list which contains both relevant & irrelevant work items (culled & unculled.) This list takes the form of a list of indices to use as a lookup table. In a threadgroup context this allows the lowest-ID threads to directly access all relevant work items. This process is somewhat complicated by the necessity of inter-wave communication, which is handled by using groupshared memory and group synchronization points.

# Known Issues

Disabling optimizations (-Od) causes CullMeshletMS.hlsl to be broken on PC. This is due to a bug in the shader compiler (dxc.exe) in the version that ships with the Windows SDK (10.0.19041.) This issue has since been fixed in the latest release available on [GitHub](https://github.com/microsoft/DirectXShaderCompiler).

# Update history

10/31/2019 – Sample creation.

2/24/2020 – Added LOD cycling & debug camera view frustum visualization.

4/28/2020 - Updated to use the D3DX12 helpers for mesh shader pipeline creation

9/2/2021 – Added note about being broken on PC when disabling optimizations using Windows SDK dxc

# Privacy Statement

When compiling and running a sample, the file name of the sample executable will be sent to Microsoft to help track sample usage. To opt-out of this data collection, you can remove the block of code in Main.cpp labeled “Sample Usage Telemetry”.

For more information about Microsoft’s privacy policies in general, see the [Microsoft Privacy Statement](https://privacy.microsoft.com/en-us/privacystatement/).