

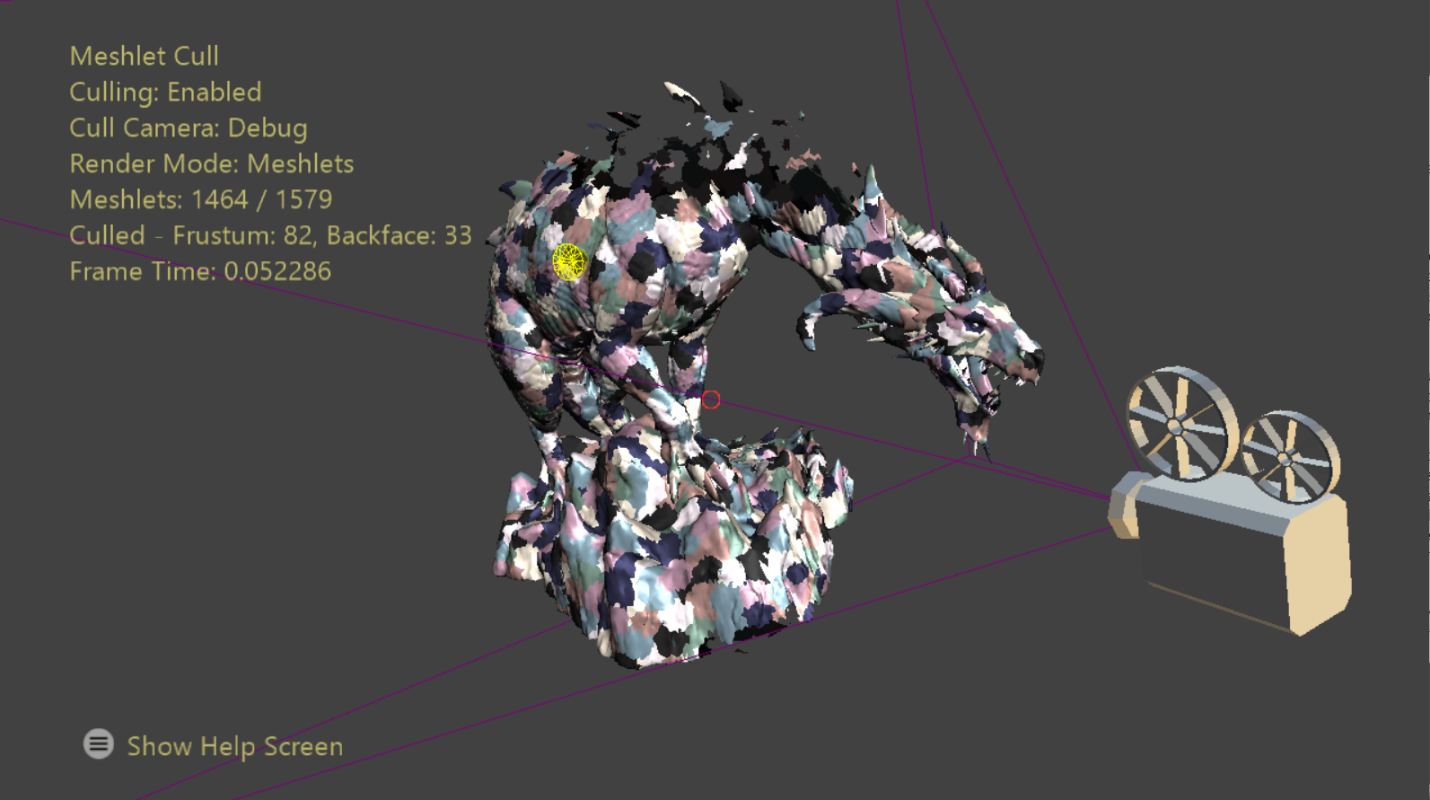
Meshlet Culling Sample

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

# Description

Amplification shaders are an optional stage preceding the mesh shader stage in the Mesh Shader Pipeline. Its purpose is to determine the required number of mesh shader threadgroups necessary in a specific GPU task, optionally passing a payload of data to its dispatched MS child threadgroups. This can be used to reduce or expand workloads before they reach the mesh shader stage in the pipeline.

This sample demonstrates how to leverage amplification shaders to cull meshlets against the camera using per-meshlet culling metadata. The goal is to minimize the number of mesh shader threadgroups required to only those portions that are deemed potentially visible before they’re ever launched.



# Building the sample

If using a Xbox Series X|S devkit, 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.Deskop.x64.

This sample does not support Xbox One.

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

# Using the sample

The focal point of the sample is showcasing amplification shader-based meshlet culling techniques. Culling can be toggled on/off, and the camera against which culling occurs may be switched between the main camera and a debug camera. The position & orientation of the debug camera is represented by the camera model in world space. The view frustum of the culling camera is visualized to more easily observe where frustum culling should occur.

There are a couple render modes provided by the sample: flat shading and a meshlet visualization. Within the meshlet view mode you can pick a meshlet from the scene to visualize its bounding sphere and a cone which represents the breadth of normals within the meshlet. No cone is rendered if that meshlet’s cone of normals forms a degenerate cone (wider than a hemisphere.) The user may also cycle between six levels-of-detail of the mesh for some variety.

Statistics are drawn on screen which presents the elapsed GPU time over scene object rendering and the number of meshlets culled from the frame.

# Controls

|  |  |  |
| --- | --- | --- |
| Action | Gamepad | Keyboard |
| Move Camera | Left Thumbstick/DPad | WASD or Arrow Keys |
| Rotate Camera | Right Thumbstick | Hold LMB + Mouse |
| Reset Camera | Right Thumbstick (Push) | N/A |
| Control Debug Camera | Left Shoulder | Left Shift |
| Toggle Culling | A Button | Tab |
| Change Render Mode | X Button | Spacebar |
| Switch Cull Camera | B Button | Q |
| Pick Meshlet | Y Button | RMB |
| Cycle LODs | Right Shoulder/Trigger | +/- |
| Show Help Menu | Menu Button | N/A |
| Exit | View Button | Escape |

# Implementation notes

This technique starts with the generation of culling data at meshlet generation time. After the meshlet list is complete each meshlet’s object space bounding sphere and normal cone is computed. Sample code of the algorithms which perform this series of steps is provided via a separate sample, MeshletConverter. This data is then compressed through quantization to save memory on-disk and read bandwidth.

A normal cone represents the *spread* of the normals within a meshlet – a cone enveloping all the normals of its primitives. It’s stored as a 4-tuple of normalized float values – a 3D unit vector representing the average normal direction and a scalar representing the dot product between that vector and the most divergent triangle normal.

For the benefit of the culling test the actual stored value is -cos(*maximal angle* + 90º). 90º are added since a triangle is observable over an entire hemisphere (or 90º about its normal) and negated as the inverted cone encompasses the set of view directions from which this surface normal would be backfacing. This reduces the culling test to a single dot product and floating-point comparison.

The amplification shader is structured to process a single meshlet on each thread. Thus, to render a mesh composed of *n* meshlets, threadgroups must be dispatched. A threadgroup size equal to the platform’s wave size is chosen to facilitate the use of wave intrinsics for threadgroup-wide operations. Each thread is responsible for performing view frustum and normal cone culling tests against its meshlet (indexed by its dispatch thread ID). Using the prefix sum wave intrinsic the indices of visible meshlets are compacted into a groupshared memory lookup table for the launched mesh shader threadgroups. The shader ends with a call to the amplification shader intrinsic function DispatchMesh to launch the requisite number of mesh shader threadgroups and specify the groupshared lookup table as payload data.

# Update history

4/20/2020 – Sample creation.

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

# 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/).