

Meshlet Instancing Sample

*This sample is compatible with the Microsoft Game Development Kit (October 2021) and Windows 10 (Version 2004) May 2020 Update*

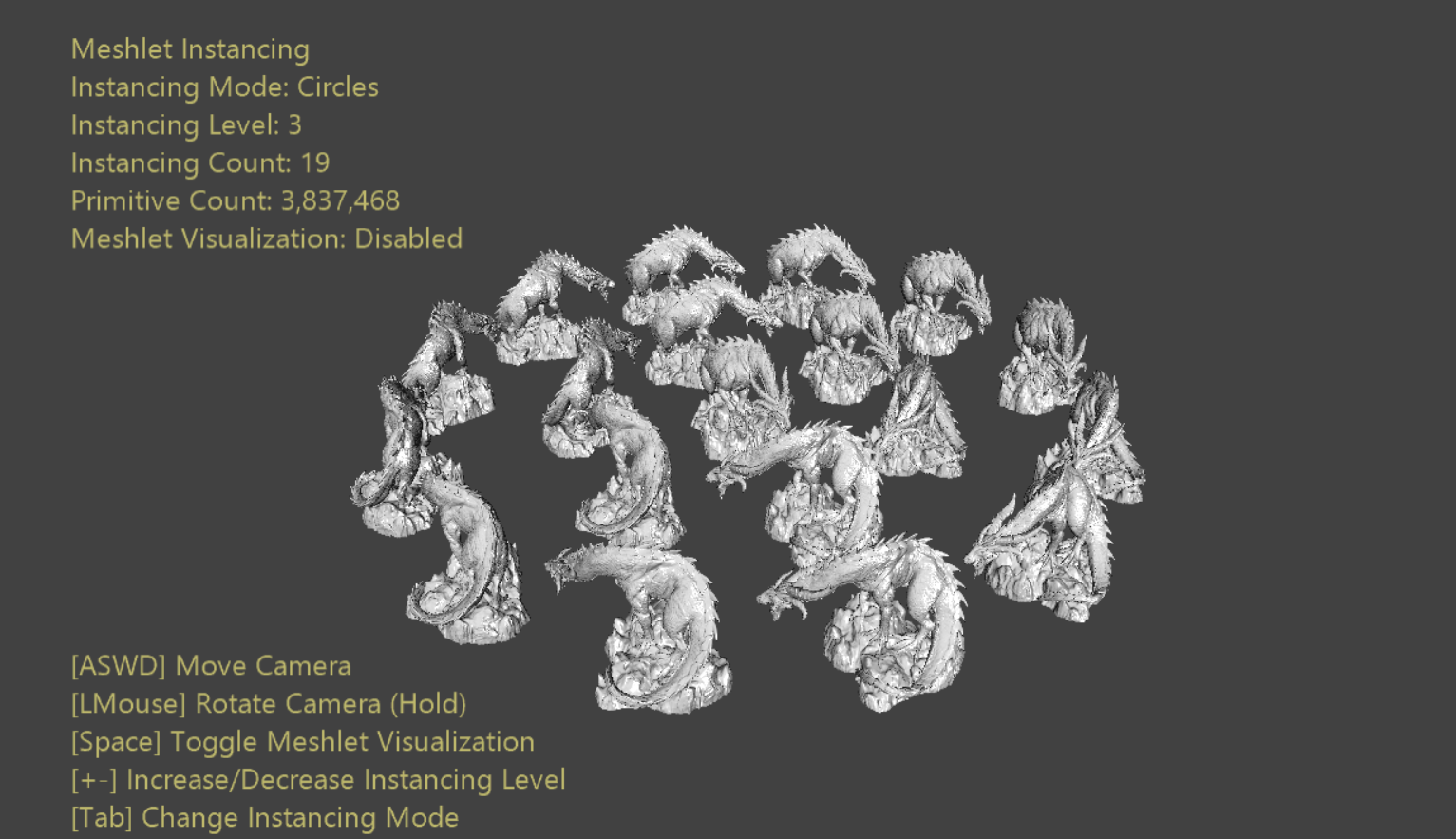
# Description

The mesh shader pipeline foregoes the input assembler which is responsible for performing index processing and instancing functionality. For this reason, the mesh shader pipeline does not expose a high-level interface for instancing as in the traditional graphics pipeline – instead the developer is responsible for using mesh shader thread IDs to implement their own instancing solution.

GPUs dispatch workloads in fixed-size chunks of threads called *waves*. This size is architecture-specific but is the minimum amount of threads which can be dispatched. One feature that the input assembler performed was packing these waves with work to optimize their thread utilization*. Wave utilization* is the ratio of wave threads which are processing actual data vs. the total number of dispatched waves.

Since the mesh shader pipeline omits the use of the input assembler, this concern of packing waves with work is lain upon the developer. Meshlets are a tool to preprocess meshes into friendly-sized chunks of work which optimize wave utilization. However, there’s no hard constraint on meshlets that they be completely full, thus leading to sub-optimal wave utilization. Specifically, the last meshlet of a mesh will most likely not be full (as the mesh was exhausted of primitives before it could be filled.) This presents an ever-growing issue as the number of meshlets in a mesh approach zero. Heavy instancing of these small meshes is common, such as in foliage, hair, particles, etc.

This sample presents a generalized method of efficiently instancing by packing many instances of the last unfilled meshlet into a single threadgroup. This minimizes the number of dead threads being dispatched keeping our wave utilization very close to 100%.



# 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 or Windows 11 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 sample provides options to choose between two instancing layouts: concentric circles & a cube. This allows the ability to scale at different rates. You may also toggle between flat shading or visualizing the underlying meshlet structure of the instances.

# Controls

|  |  |  |
| --- | --- | --- |
| Action | Gamepad | Keyboard |
| Move Camera | Left Thumbstick | WASD or Arrow Keys |
| Rotate Camera | Right Thumbstick | Hold LMB + Mouse |
| Reset camera | Right Thumbstick (Push) | - |
| Change Instancing Mode | A | Tab |
| Toggle Meshlet Visualization | X | Spacebar |
| Increase Instancing Level | Right Shoulder | + |
| Decrease Instancing Level | Right Trigger | - |
| Exit | View Button | Escape |

# Implementation notes

Mesh shader instancing is a simple matter of dispatching enough shader instances to complete the work and using supplied IDs to determine the correct meshlet & instance to process. The chosen indexing scheme determines which threadgroups process which bit of work.

Assuming one meshlet per threadgroup, the most straightforward implementation would be to dispatch MeshletCount \* InstanceCount threadgroups. We can be certain this is enough threadgroups to fully process our workload. This also provides a very straightforward indexing scheme:

* *MeshletIndex* = *GroupID.x* / *InstanceCount*
* *InstanceIndex* = *GroupID.x* % *InstanceCount*

This means the first *InstanceCount* threadgroups will all process Meshlet *0*, but different instances. More importantly the last *InstanceCount* threadgroups will all process Meshlet *(MeshletCount – 1)*. We assume the first *(MeshletCount – 1)* meshlets are ‘full’ (having near maximum vertices and/or primitives) and the last meshlet is less filled. The last *(MeshletCount – 1)* threadgroups will thus have less-optimal wave utilization than the first *(MeshletCount – 1) \* InstanceCount* threadgroups. If sparse enough multiple instances could be packed into a single threadgroup.

In fact, assuming the threadgroup size is equal to the maximum meshlet size we can fit instances into a single threadgroup. Thus we need packed threadgroups to process the last meshlet. This provides optimal wave efficiency at the cost of some extra ALU of these threadgroups in the shader.

# Update history

10/31/2019 – 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/).