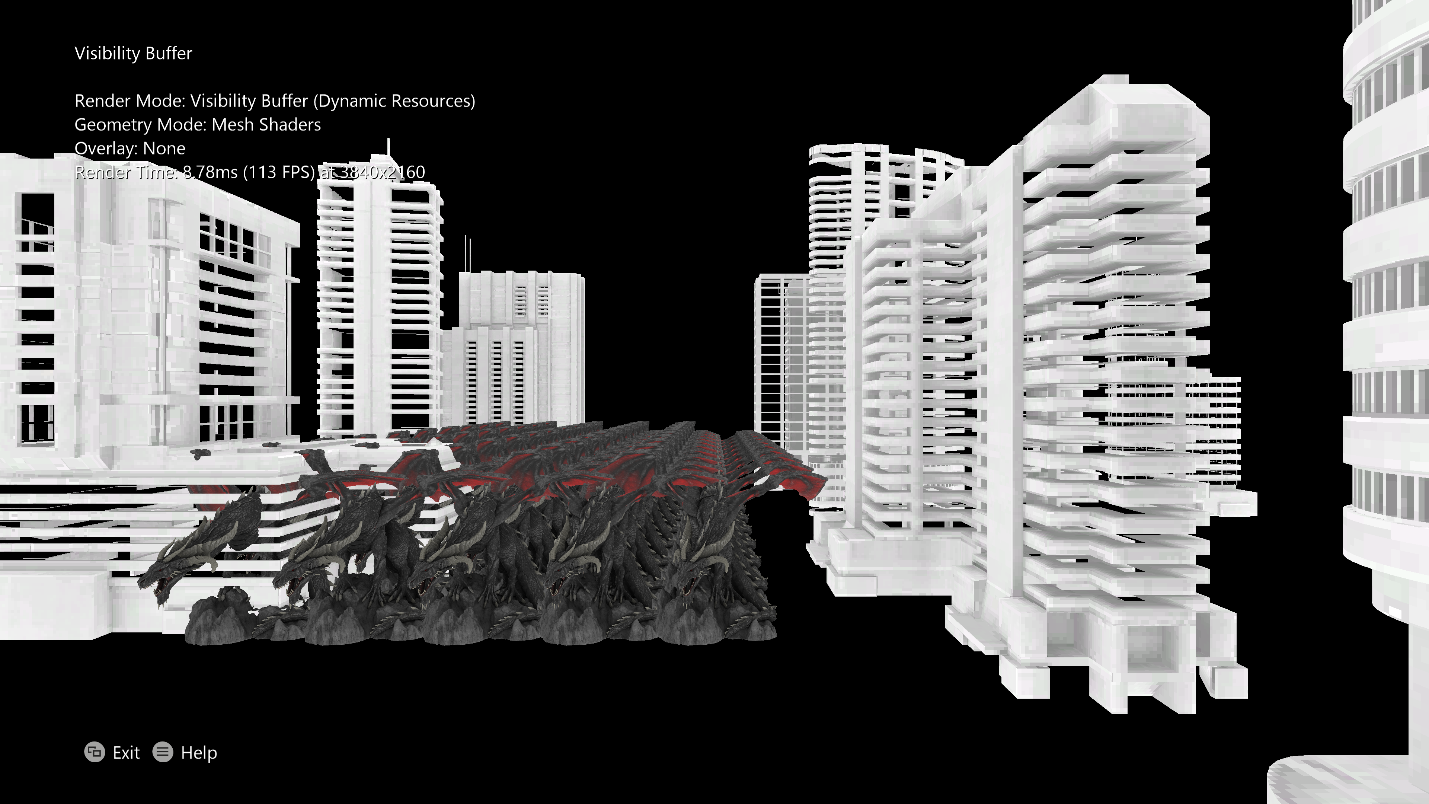
Visibility Buffer Sample

*This sample is compatible with the Microsoft Game Development Kit (April 2021)*

# Description

This sample demonstrates a visibility buffer (deferred) rendering technique, making use of Mesh Shaders and HLSL 6.6 Dynamic Resources.



# Building the sample

If using an Xbox One devkit, set the active solution platform to Gaming.Xbox.XboxOne.x64.

If using Xbox Series X|S, set the active solution platform to Gaming.Xbox.Scarlett.x64.

Building for PC (Gaming.Desktop.x64) requires the [DirectX Agility SDK](https://devblogs.microsoft.com/directx/gettingstarted-dx12agility/) due to the usage of HLSL SM 6.6 features. The Agility SDK is included as a NuGet package in the sample. It also makes use of Microsoft.Windows.SDK.CPP NuGet packages to get the latest Windows SDK (22000) version of the DXC.exe compiler. Developers can also use the latest DXC directly from [Github](https://github.com/microsoft/DirectXShaderCompiler/releases).

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

# Using the sample

The sample features a flying / first-person camera to allow the camera to be placed anywhere within the scene. Full explanation of the controls are listed below:

# Controls

|  |  |
| --- | --- |
| Action | Gamepad |
| View Controls | Start Button |
| Toggle between Visibility Buffer (Deferred) Rendering, and Forward Rendering. Visibility Buffer Rendering uses dynamic resources. Forward does not. | A |
| Cycle Visibility Buffer overlay. Shows primitive IDs or object IDs as overlay. Internally both are rendered to a single 32-bit UINT render target. | B |
| Toggle Vertex Shader / Mesh Shader geometry rendering. Applies only to Visibility Buffer mode. Only Vertex Shaders are supported on Xbox One. | X |
| Camera Forwards / Backwards / Strafe | Left stick |
| Camera Yaw / Pitch | Right stick |
| Camera Up / Down | Left / Right Trigger |
| Fast flying mode (Turbo) | Click + Hold Left stick |
| Quit | View Button |

# Implementation notes

Two render paths are shown (toggled by pressing A).

In the Forward path, typical rasterization is performed, and all resources (Index Buffers, Vertex Buffers, Textures, Samplers, etc.) are bound either through the root signature or to the Input Assembler.

In the Visibility Buffer path, “deferred” rasterization is performed, with the option of Mesh Shaders or Vertex Shaders for geometry shading, and a Compute Shader performing the final scene shading. If using Mesh Shaders, all resources are accessed “bindlessly” through the HLSL 6.6 features ResourceDescriptorHeap[] and SamplerDescriptorHeap[]. While in the Vertex Shader mode, the index buffer and vertex buffer are bound to the Input Assembler for performance reasons.

In the Vertex Shader mode, it is also possible to only bind the index buffer, and access the vertex buffer through the ResourceDescriptorHeap[], however this caused significant performance losses on some hardware.

Mesh Shaders are not supported on Xbox One, so only the Vertex Shader mode is available.

Visibility Buffer

The Visibility Buffer path is an implementation of a “deferred” renderer. However the initial rasterization path only outputs a single 32-bit UINT render target (Visibility Buffer), containing an object identifier in the first 12 bits (4096 possible objects) and a primitive identifier in the last 20 bits (1,048,576 triangles per object). Then, a compute shader pass consumes the Visibility Buffer, loads per-object constant information from the ResourceDescriptorHeap, and then loads Index Buffers / Vertex Buffers / Textures from the ResourceDescriptorHeap. The compute shader then reconstructs the hit point of each triangle and determines the appropriate interpolated texture coordinates and screen-space derivatives. Note that this is unlike a normal deferred renderer, where this data would have been stored in intermediate render targets after the first pass. Finally, a SamplerState object is loaded using the SamplerDescriptorHeap, and the texture is sampled and output to the screen.

Geometry Shading

This sample implements both vertex shaders and mesh shaders (where supported). The visibility buffer must include the primitive ID, which can be taken from the SV\_PrimitiveID system semantic after a vertex shader pass, or passed as per-primitive output from a mesh shader.

For mesh shaders, a simple launch-time meshlet splitting algorithm is performed. This algorithm expects a pre-optimized index buffer (such as the output of DirectXMesh’s MeshConvert with the -op flag). It then splits the index buffer into meshlets that meet a max vertex / max primitive limit, while maintaining the existing primitive ordering. Then a very simple mesh shader can load the meshlet information through ResourceDescriptorHeap[], perform normal vertex shading, and output the primitiveID in a per-primitive export. This ID maintains the same ordering as the original index buffer, so it can be used with the same compute shader, without the compute shader needing to understand the meshlet format.

Performance

The performance of this technique depends on many factors. The geometry shading costs and tradeoffs between mesh shaders and vertex shaders will depend on the target GPU platform and asset sizes.

Additionally, the Forward Renderer’s pixel shader work can be run in parallel with the vertex shader work, while the Visibility Buffer Renderer must perform it’s “pixel shading” in compute, after all vertices have been processed. This means that unless the pixel work outweighs the vertex work, any overdraw costs will be hidden by the larger vertex shader work.

However, the compute shader is very efficient, and if pixel shader work dominates, the Visibility Buffer Renderer can become more performant. This is primarily due to the complete lack of overdraw and the slight performance wins of mesh shaders, however there may also be benefits to avoiding quad helper lanes, and quirks of primitive output which may slow down pixel shader waves.

In this sample both the Forward Renderer and Visibility Buffer Renderer perform many redundant texture fetches in a loop at the end of their shading. This is in attempt to increase the pixel shader costs to better reflect overdraw performance differences, as without this the vertex shader work always dominates.

Finally, although no traditional Deferred Renderer is implemented in this sample, there may be benefits to this method over that as well. A key note is far lower render target memory requirements, with the Visibility Buffer only being a single 32-bit target, as opposed to multiple targets for diffuse/normal/specularity/etc. Similarly there is much less fill-rate required, as only the one target is written in the initial pass, and only one UAV is output from the compute shader. Required resources are pulled directly from the descriptor heap when necessary.

# 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/).