

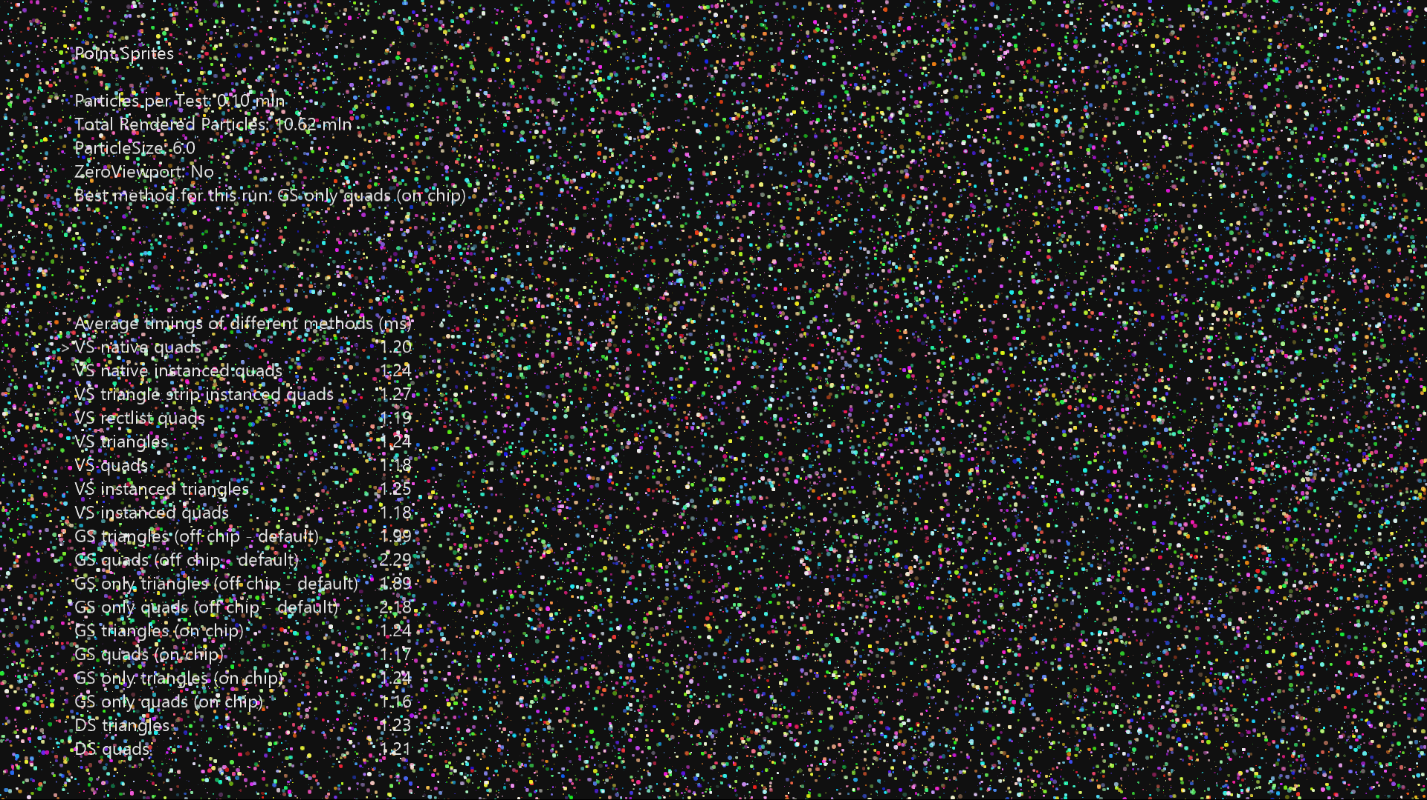
Point Sprites

*This sample is compatible with the Microsoft Game Development Kit (June 2020)*

# Description

Demonstrates ten methods of rendering point sprites in DirectX 12. Each method is profiled, and a heads-up comparison is displayed to show performance characters of each method given the rendering parameters.

On Xbox One, the sample also details how to specify on/off-chip memory for geometry shaders. This distinction isn’t available on Xbox Series X|S.



# Using the sample

This sample uses the following controls.

|  |  |
| --- | --- |
| Action | Gamepad |
| Exit the sample. | View Button |
| Increase Sprite Size | A Button |
| Decrease Sprite Size | B Button |
| Toggle Null Viewport | Y Button |
| Change Selected Test | D-Pad Up/Down |
| Toggle Highlighted Test | X Button |

# Implementation notes

As the graphics pipeline evolved, first-class support for point sprites has been removed in favor of using fully flexible and customizable stages like the Geometry Shader. In fact, point sprites can be implemented in every geometry-related stage of the GPU – vertex, geometry, tessellation, and mesh shaders. Although the ideal choice of the method should be based on measuring the performance on a case-by-case basis, it appears that using Geometry Shader for expanding point sprites is not the most efficient way to render simple point sprites.

## Triangles vs Quads

Depending on where the rendering pipeline is bottlenecked, triangles or quads may have different performance. In fill-rate bound situations, quads have the performance edge over the triangles because they don’t need to run pixel shaders on transparent pixels. When rendering is bound on geometry stages, it makes sense to output triangles instead of quads. In many complex rendering situations triangles may be faster than quads.

This sample demonstrates each method contrasting triangles and quads to show the difference at different sizes of point sprites.

## Method 1 - Vertex + Geometry Shader

The pipeline is initialized such that the vertex shader reads the vertex normally, then the geometry shader outputs either a quad or a triangle per input vertex. These tests also compare on-chip vs. off-chip geometry shader allocations. The compiler defaults to using off-chip memory which is used to store inter-stage data between the vertex and geometry stages.

Advantages

* Well known and simple.

Disadvantages

* Requires most set up.
* Not the fastest in all cases.

## Method 2 – Geometry Shader Only

This method uses an empty vertex shader. In order to load the vertex data the geometry shader performs buffer loads on a raw byte view of a vertex buffer, using the SV\_PrimitiveID index (for point lists, SV\_PrimitiveID in the geometry shader is the same as SV\_VertexID in the vertex shader). After the vertex is loaded, the point sprite expansion is performed as in Method 1. On-chip vs off-chip GS performance is similarly tested here.

Advantages

* No input layout is required as VS is a no-op.
* It performs faster than Method 1 as no internal traffic is taking place between the VS and the GS.

Disadvantages

* The manual vertex load necessitates different shader variants for each vertex layout. For highly optimized code this is not a problem.
* This method is a bit more difficult to understand.

## Method 3 – Vertex Shader Only

Using a geometry shader just to expand a point into a quad or a triangle is not really required in DX11, and a vertex shader can be used to do that instead. In DX11 the vertex shader stage can read raw byte UAVs, so having SV\_VertexID and the raw byte view of the vertex buffer, it’s possible to read the vertex manually.

So to expand the vertex into a triangle or a quad, we just need to render either 3 times more or 6 times more vertices in the draw call, perform division by 3 or 6 in the shader to get the index of the vertex, load the vertex using this index, and expand it based on the remainder of the division to get the sprite’s corner.

Advantages

* This method is simple, fast, and easy for any level graphics programmer to understand.
* No input layout is required for the VS as it only consumes a system generated value SV\_VertexID.

Disadvantages

* Vertex load is manual so a different shader may be needed for loading a different vertex layout. This isn’t a problem if you want the absolute fastest method.

## Method 4 – Vertex Shader Instancing

We can use instancing to make the GPU to load point sprite vertices for us and SV\_VertexID to determine the corner of the sprite for expansion. This method is slightly slower than the previous method, but still consistently faster than any other methods.

Advantages

* Second fastest method, very similar performance to method 3.
* No division is required so the shader has fewer ALUs.
* Vertex load is performed using input layouts so it’s possible to load different vertex layouts without modifying the shader.
* Seems to be the winning method on very large sizes of point sprites.

Disadvantages

* None

## Method 5 – Tessellation Stages

It’s possible to use the tessellation stages to generate triangles and quads from a single input vertex. This method’s performance is on par with the geometry shader-based methods, but it is more flexible as more sprite shapes are possible. For example, by tessellating in quad domain, it’s possible to output circles instead of quads – this might be more efficient if the pixel shader is very slow and the sprites are circles.

Advantages

* Can output almost arbitrary geometry shapes.

Disadvantages

* Same as for geometry shader approach.

## Method 6 – Mesh Shader Pipeline

Mesh shaders provide a very natural programming model for rendering procedural geometry. This method can see tremendous benefits from the omission of the input assembler in the mesh shader pipeline. The performance benefits are reduced as the workload becomes bottlenecked in the post-cull portions of the pipeline.

There is no option of triangles vs. quads when using mesh shaders – output geometry is an indexed vertex list of either lines or triangles.

Advantages

* Lightning fast! Also provides the ability to cull particles within the mesh shader (not implemented.)

Disadvantages

* Mesh shaders are structurally a bit more complex and require an understanding of both compute and traditional graphics workloads. The lack of input assembler necessitates a bit of computation on the CPU dispatch side as well as manual input fetching in the shader code.

## Conclusion:

All the above methods will perform differently given a different GPU load, so choose the method that works best in your case.

# Update history

4/12/2019 – Port to DX12 from Xbox Sample Framework.

2/20/2020 – Update for Xbox Series X|S

6/8/2020 – Added mesh shader particles.

# 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/).