

Simple Instancing Sample (DX12)

*This sample is compatible with the Windows 10 Anniversary Update SDK (14393)*

# Description

This sample demonstrates how to use instancing with the Direct3D 12 API.

# Using the sample



|  |  |  |
| --- | --- | --- |
| Action | Gamepad | Keyboard |
| Rotate camera | Left Thumbstick | WASD / Left Mouse and Left Mouse Button |
| Change instance count | LB / RB | Q / E |
| Center | n/a | Home |
| Reset simulation | A | Space |
| Exit | View Button | Esc |
| Action | Gamepad | Keyboard |
| Rotate camera | Left Thumbstick | WASD / Left Mouse and Left Mouse Button |
| Change instance count | LB / RB | Q / E |
| Center | n/a | Home |
| Reset simulation | A | Space |
| Exit | View Button | Esc |

# Implementation notes

There are four areas to consider when rendering instanced geometry:

1. **Geometric Data**

In the case of this sample, this includes a vertex and index buffer containing vertices and indices describing the faces of a cube. It also includes the *pipeline state object* required to render this cube into the world. All these components are set up and manipulated in the same manner as non-instanced geometry would be. (See *CreateDeviceDependentResources* in SimpleInstancing12.cpp)

1. **Instance Data**

For standard D3D12 instanced rendering, per instance data is provided via one or more vertex buffers. These vertex buffers are created in the same way any other vertex buffer would be. This sample uses two vertex buffers. One is static and contains per-instance color data (which is unchanging for the lifetime of the sample). The other is dynamic and contains per-instance position and orientation information (changes every frame).

1. **Instancing Layout**

To render the geometry instanced, D3D requires information about how the vertex data supplied is to be interpreted. This is done using an array of *D3D12\_INPUT\_ELEMENT\_DESC* structures, much the same way that standard rendering is done. However, extra elements are added to this structure. Geometric data is flagged with the *D3D12\_INPUT\_CLASSIFICATION\_PER\_VERTEX\_DATA* value for the *InputSlotClass* element, as usual, but per-instance data uses the *D3D12\_INPUT\_CLASSIFICATION\_PER\_INSTANCE\_DATA* value. The *InputSlot* element is also used to denote the vertex stream that each piece of data is pulled from.

The vertex shader uses a vertex structure that’s defined as if the geometric and per-instance data were all lumped together (that reflects the layout described in the *D3D12\_INPUT\_ELEMENT\_DESC* array).

***Note:*** *This sample uses the* D3D12\_APPEND\_ALIGNED\_ELEMENT *constant for the* AlignedByteOffset *element to automatically align data correctly in the input layout. This only works if the structure of the vertex buffer in question contains correctly aligned data. If you are skipping (or ignoring) elements within your vertex data, then exact alignment offsets will be required.*

1. **Rendering**

Rendering instanced data is simple once the previous points are nailed down. The *ID3D12GraphicsCommandList::IASetVertexBuffers* API is used to set the vertex buffers used as input (in this case, the input buffers), and the *ID3D12GraphicsCommandList::DrawIndexedInstanced* API is used to render. The rest of the rendering setup is performed the same as for standard non-instanced rendering.

# Update history

Initial release August 2016

# 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/).