Instancing Sample

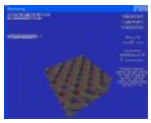
See Also

Collapse All

This sample demonstrates the instancing feature available with Direct3D 9. A vs_3_0 device is required for the hardware version of this feature. The sample also shows alternate ways of achieving results similar to hardware instancing, but for adapters that do not support vs_3_0. The shader instancing technique shows the benefits of efficient batching of primitives.

Note

On graphics hardware that does not support vs_2_0, the sample will run as a reference device.



Path

Source	*SDK root*\Samples\C++\Direct3D\Instancing
Executable	*SDK root*\Samples\C++\Direct3D\Bin*x86 or x64*\Instancing.exe

How the Sample Works

The sample demonstrates four different rendering techniques to achieve the same result: to render many nearly identical boxes (objects with small numbers of polygons) in the scene. The boxes differ by their position and color.

The user can vary the number of boxes in the scene between one and 1000. Use the sample to monitor performance as the number of boxes increases. As you change the number of boxes, the vertex and index buffer resources are recreated by OnCreateBuffers and OnDestroyBuffers.

Technique 1: Hardware Instancing

Hardware instancing requires a vs_3_0-capable device. The instance-specific data is stored in a second vertex buffer. The rendering is implemented in the sample application's OnRenderHWInstancing method. **IDirect3DDevice9::SetStreamSourceFreq** is used with D3DSTREAMSOURCE_INDEXEDDATA to specify the number of boxes and with D3DSTREAMSOURCE_INSTANCEDATA to specify the frequency of the instance data (in this case, frequency equals one).

Drawing the scene is accomplished as follows:

```
pd3dDevice->DrawIndexedPrimitive( D3DPT_TRIANGLELIST, 0, 0, 4 * 6, 0, 6 * 2 );
```

Technique 2: Shader Instancing (with Draw Call Batching)

This is the most efficient technique that does not use the hardware to perform the instancing. One call to **IDirect3DDevice9::DrawIndexedPrimitive** handles multiple primitives at once. The instance data is stored in a system memory array, which is then copied into the vertex shader's constants at draw time. Since vs_2_0 only guarantees 256 constants, it is not possible for the entire array of instance data to fit at once. This sample's .fx file can only batch-process 120 constants (one float4 is required for box position, and one float4 is required for box color). Rendering is performed in the sample's OnRenderShaderInstancing method.

```
int nRenderBoxes = min( nRemainingBoxes, g_nNumBatchInstance );
...
pd3dDevice->DrawIndexedPrimitive( D3DPT_TRIANGLELIST, 0, 0, nRenderBoxes * 4 * 6, 0, nRenderBoxes * 6 * 2 );
```

This method would also work on a vs_1_1 part, although not as efficiently because vs_1_1 only guarantees 96 constants. This means that batching is nearly three times more efficient on vs_2_0 hardware than on vs_1_1 hardware, but even on vs_1_1 hardware, batching still outperforms no batching.

Technique 3: Constants Instancing (without Draw Call Batching)

This technique is somewhat similar to Technique 2 without the batching (batch size = 1). It will render one box at a time, by first setting its position and color and then calling DrawIndexedPrimitive. The technique is implemented in the sample's OnRenderConstantsInstancing method.

Technique 4: Stream Instancing

As in Technique 1, this technique uses a second vertex buffer to store instance data. Instead of setting vertex shader constants to update the position and color for each box, this technique changes the offset in the instance stream buffer and draws one box at a time. The technique is implemented in the sample's OnRenderStreamInstancing method.

CPU- vs. GPU-bound

When the sample renders a large number of boxes, only Techniques 1 and 2 are GPU-bound, while Techniques 3 and 4 waste many CPU cycles by making numerous calls per frame to DrawIndexedPrimitive.

The performance numbers for rendering a large number of boxes show that only the first two techniques are GPU-bound, while the last two techniques waste a lot of CPU cycles by making numerous calls to DrawIndexedPrimitive per frame. To better illustrate the penalty from being CPU-bound in a real world application, the sample has the Goal option which simulates a secondary task (in a game, this could be AI or physics) competing for CPU. The sample queries the time elapsed since the last frame and if there is time remaining

inside the goal time/frame, the sample allows time to pass to represent game logic. It reports the amount of time used in the Remaining for logic statistic. This is the percentage of CPU cycles that are available to spend on non-rendering algorithms such as game logic or physics; the higher the percentage, the better. If the scene is loaded with 1000 boxes, it was observed that the efficient techniques have 2x better CPU efficiency than the CPU-bound techniques. In other words, using Techniques 3 and 4 to render many instances will starve other tasks of CPU time.

The following table summarizes hardware and memory requirements for different rendering techniques:

Rendering Technique	Requires vs_3_0 Card	CPU-bound	Requires Additional Vertex Buffer	Requires Additional Shader Constant Buffer
Hardware Instancing	Yes	No	Yes	No
Shader Instancing	No	No	No	Yes
Constants Instancing	No	Yes	No	Yes
Stream Instancing	No	Yes	Yes	No

See Also

Efficiently Drawing Multiple Instances of Geometry (Direct3D 9)