Instancing10 Sample

Collapse All

This sample demonstrates the use of the instancing and texture arrays to reduce the number of draw calls required to render a complex scene. In addition, it uses AlphaToCoverage to avoid sorting semi-transparent primitives.

Path

Source	*SDK root*\Samples\C++\Direct3D10\Instancing10
Executable	*SDK root*\Samples\C++\Direct3D10\Bin*x86 or x64*\Instancing10.exe

How the Sample Works

Reducing the number of draw calls made in any given frame is one way to improve graphics performance for a 3D application. The need for multiple draw calls in a scene arises from the different states required by different parts of the scene. These states often include matrices and material properties. One way to combat these issues is to use Instancing and Texture Arrays. In this sample, instancing enables the application to draw the same object multiple times in multiple places without the need for the CPU to update the world matrix for each object. Texture arrays allow multiple textures to be loaded into same resource and to be indexed by an extra texture coordinate, thus eliminating the need to change texture resources when a new object is drawn.

The sample draws several trees, each with many leaves, and several blades of grass using 3 draw calls. To achieve this, the sample uses one tree mesh, one leaf mesh, and one blade mesh instanced many time throughout the scene and drawn with DrawIndexedInstanced. To achieve variation in the leaf and grass appearance, texture arrays are used to hold different textures for both the leaf and grass instances. AlphaToCoverage allows the sample to further unburden the CPU and draw the leaves and blades of grass in no particular order. The rest of the environment is drawn in 6 draw calls.

Instancing the Island and the Tree

In order to replicate an island or a tree the sample needs two pieces of information. The first is the mesh information. In this case, the tree mesh is loaded from tree.sdkmesh and the island mesh is loaded from island.sdkmesh. The second piece of information is a buffer containing a list of matrices that describe the locations of all tree instances. In this case, an array of 4x4 D3DMATRIX structures is defined and translated using random position and scaling data with the sample's CreateRandomTreeMatrices()function.

The sample uses IASetVertexBuffers to bind the mesh information to vertex stream 0 and the matrices to stream 1:

```
ID3D10Buffer* pVB[2];
pVB[0] = g_MeshTree.GetVB10( 0, 0 );
pVB[1] = g_pTreeInstanceData;
Strides[0] = ( UINT )g_MeshTree.GetVertexStride( 0, 0 );
Strides[1] = sizeof( D3DXMATRIX );
pd3dDevice->IASetVertexBuffers( 0, 2, pVB, Strides, Offsets );
```

To get this information into the shader correctly, the following InputLayout is used:

```
const D3D10_INPUT_ELEMENT_DESC instlayout[] =
{
    { L"POSITION", 0, DXGI_FORMAT_R32G32B32_FLOAT, 0, 0, D3D10_INPUT_PER_VERTEX_DATA, 0 },
    { L"NORMAL", 0, DXGI_FORMAT_R32G32B32_FLOAT, 0, 12, D3D10_INPUT_PER_VERTEX_DATA, 0 },
    { L"TEXTURE0", 0, DXGI_FORMAT_R32G32_FLOAT, 0, 24, D3D10_INPUT_PER_VERTEX_DATA, 0 },
    { L"mTransform", 0, DXGI_FORMAT_R32G32B32A32_FLOAT, 1, 0, D3D10_INPUT_PER_INSTANCE_DATA, 1 },
    { L"mTransform", 1, DXGI_FORMAT_R32G32B32A32_FLOAT, 1, 16, D3D10_INPUT_PER_INSTANCE_DATA, 1 },
    { L"mTransform", 2, DXGI_FORMAT_R32G32B32A32_FLOAT, 1, 32, D3D10_INPUT_PER_INSTANCE_DATA, 1 },
    { L"mTransform", 3, DXGI_FORMAT_R32G32B32A32_FLOAT, 1, 48, D3D10_INPUT_PER_INSTANCE_DATA, 1 },
};
```

The vertex shader will be called (number of vertices in mesh)*(number of instance matrices) times. Because the matrix is a shader input, the shader can position the vertex at the correct location according to which instance it happens to be processing.

Instancing the Leaves

Because one leaf is instanced over an entire tree and one tree is instanced several times throughout the sample, the leaves must be handled differently than the tree and grass meshes. The matrices for the trees are loaded into a constant buffer. The InputLayout is setup to make sure the shader sees the leaf mesh data m_iNumTreeInstances time before stepping to the next leaf matrix. The last element, fOcc, is a baked occlusion term used to shade the leaves.

```
const D3D10_INPUT_ELEMENT_DESC leaflayout[] =
{
    { L"POSITION", 0, DXGI_FORMAT_R32G32B32_FLOAT, 0, 0, D3D10_INPUT_PER_VERTEX_DATA, 0 },
    { L"TEXTURE0", 0, DXGI_FORMAT_R32G32_FLOAT, 0, 12, D3D10_INPUT_PER_VERTEX_DATA, 0 },
    { L"mTransform", 0, DXGI_FORMAT_R32G32B32A32_FLOAT, 1, 0, D3D10_INPUT_PER_INSTANCE_DATA, m_iNumTreeInstances },
    { L"mTransform", 1, DXGI_FORMAT_R32G32B32A32_FLOAT, 1, 16, D3D10_INPUT_PER_INSTANCE_DATA, m_iNumTreeInstances },
    { L"mTransform", 2, DXGI_FORMAT_R32G32B32A32_FLOAT, 1, 32, D3D10_INPUT_PER_INSTANCE_DATA, m_iNumTreeInstances },
    { L"mTransform", 3, DXGI_FORMAT_R32G32B32A32_FLOAT, 1, 48, D3D10_INPUT_PER_INSTANCE_DATA, m_iNumTreeInstances },
    { L"fOcc", 0, DXGI_FORMAT_R32_FLOAT, 1, 64, D3D10_INPUT_PER_INSTANCE_DATA, m_iNumTreeInstances },
};
```

The input assembler automatically generates an InstanceID which can be passed into the shader. The following snippet of shader code

demonstrates how the leaves are positioned.

```
int iTree = input.InstanceId%g_iNumTrees;
float4 vInstancePos = mul( float4(input.pos, 1), input.mTransform  );
float4 InstancePosition = mul(vInstancePos, g_mTreeMatrices[iTree] );
```

If there were 3 trees in the scene, the leaves would be drawn in the following order: Tree1, leaf1; Tree2, leaf1; Tree3, leaf1; Tree1, leaf2; Tree2, leaf2; etc...

Instancing the Grass

Grass rendering is handled differently than the Tree and Leaves. Instead of using the input assembler to instance the grass using a separate stream of matrices, the grass is dynamically generated in the geometry shader. The top of the island mesh is passed to the vertex shader, which passes this information directly to the GSGrassmain geometry shader. Depending on the grass density specified, the GSGrassmain calculates psuedo-random positions on the current triangle that correspond to grass positions. These positions are then passed to a helper function that creates a blade of grass at the point. A 1D texture of random floating point values is used to provide the psuedo-random numbers. It is indexed by vertex ids of the input mesh. This ensures that the *random* distribution doesn't change from frame to frame.

Alternatively, the grass can be rendered in the same way as the tree by placing the vertices of a quad into the first vertex stream and the matrices for each blade of grass in the second vertex stream. The fOcc element of the second stream can be used to place precalculated shadows on the blades of grass (just as it is used to precalculate shadows on the leaves). However, the storage space for a stream of several hundred thousand matrices is a concern even on modern graphics hardware. The grass generation method of using the geometry shader, while lacking built-in shadows, uses far less storage.

Changing Leaves with Texture Arrays

Texture arrays are just what the name implies. They are arrays of textures, each with full mipmaps. For a texture2D array, the array is indexed by the z coordinate. Because the InstanceID is passed into the shader, the sample uses InstanceID%numArrayIndices to determine which texture in the array to use for rendering that specific leaf or blade of grass.

Drawing Transparent Objects with Alpha To Coverage

The number of transparent leave and blades of grass in the sample makes sorting these objects on the CPU expensive. Alpha to coverage helps solve this problem by allowing the Instancing sample to produce convincing results without the need to sort leaves and grass back to front. Alpha to coverage must be used with multisample anti-aliasing (MSAA). MSAA is a method to get edge anti-aliasing by evaluating triangle coverage at a higher-frequency on a higher resolution z-buffer. With alpha to coverage, the MSAA mechanism can be tricked into creating psuedo order independent transparency. Alpha to coverage generates a MSAA coverage mask for a pixel based upon the pixel shader output alpha. That result is combined by AND with the coverage mask for the triangle. This process is similar to screen-door transparency, but at the MSAA level.

Alpha to coverage is not designed for true order independent transparency like windows, but works great for cases where alpha is being used to represent coverage, like in a mipmapped leaf texture.