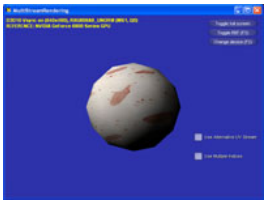
MultiStreamRendering Sample

Collapse All



Path

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

Sample Overview

This sample explains how to perform multi-stream rendering on Direct3D 9 and Direct3D 10 platforms. Multi-stream rendering allows vertex attributes to be split between different vertex streams. Multiple streams are referenced using a single index buffer. In addition, Direct3D 10 allows multi-stream and multi-index rendering, where multiple indices are used to access vertex data that is stored at frequencies. In this sample, the Direct3D 10 codepath references positions that are stored once per vertex and, in the same shader, references normals that are only stored once per triangle.

Multi-Stream Single-Index Rendering in Direct3D 9

Multi-Stream Single-Index rendering renders geometry using multiple vertex streams that are indexed by the same index buffer. This means that all data must be stored at the same frequency (namely, data is stored per-vertex). For mult-stream single-index rendering, 4 input vertex buffers are created. They store position, normal, textures coordinates, and alternative texture coordinates respectively.

The first step is to create a vertex declaration that incorporates all of these individual vertex streams. The goal of this vertex declaration is to make the multiple individual streams look like one unified stream to the vertex shader.

```
// Create a Vertex Decl for the MultiStream data.
// Notice that the first parameter is the stream index.  This indicates the
// VB that the particular data comes from.  In this case, position data
// comes from stream 0.  Normal data comes from stream 1.  Texture coordinate
// data comes from stream 2.
D3DVERTEXELEMENT9 declDesc[] =
{
    {0, 0, D3DDECLTYPE_FLOAT3, D3DDECLMETHOD_DEFAULT, D3DDECLUSAGE_POSITION, 0},
    {1, 0, D3DDECLTYPE_FLOAT3, D3DDECLMETHOD_DEFAULT,  D3DDECLUSAGE_NORMAL, 0},
    {2, 0, D3DDECLTYPE_FLOAT2, D3DDECLMETHOD_DEFAULT,  D3DDECLUSAGE_TEXCOORD, 0},
    {0xFF,0,D3DDECLTYPE_UNUSED, 0,0,0}// D3DDECL_END
};
pd3dDevice->CreateVertexDeclaration( declDesc, &g_pDecl );
```

Before rendering the sample must make sure that all streams are bound, that the index buffer is bound, and that the correct vertex declaration is set. Notice that the third vertex stream (stream number 2) can be switched between the original set of texture coordinates and the alternate set of texture coordinates without affecting the other streams.

```
// Setup our multiple streams
pd3dDevice->SetStreamSource( 0, g_pVBs[ST_VERTEX_POSITION], 0, sizeof(D3DXVECTOR3) );
pd3dDevice->SetStreamSource( 1, g_pVBs[ST_VERTEX_NORMAL], 0, sizeof(D3DXVECTOR3) );
if(g_bUseAltUV)
    pd3dDevice->SetStreamSource( 2, g_pVBs[ST_VERTEX_TEXTUREUV2], 0, sizeof(D3DXVECTOR2) );
else
    pd3dDevice->SetStreamSource( 2, g_pVBs[ST_VERTEX_TEXTUREUV], 0, sizeof(D3DXVECTOR2) );

// Set our index buffer as well
pd3dDevice->SetIndices( g_pIB );

// Set A Multistream Vertex Decl insted of FVF
pd3dDevice->SetVertexDeclaration( g_pDecl );
```

This is all of the work necessary to facilitate multi-stream single-index rendering on Direct3D 9. The vertex shader is written as if all of the input came in from a single stream.

Multi-Stream Single-Index Rendering in Direct3D 10

The setup is similar for performing multi-stream single-index rendering in Direct3D 10. The same vertex buffers are created (position, normal, texture coordinates, alternate texture coordinates). The sample then creates and input layout incorporating the separate vertex streams. Again, the purpose is to make the multiple vertex streams look like one stream to the vertex shader.

```
// Create a Input Layout for the MultiStream data.
// Notice that the 4th parameter is the stream index.  This indicates the
// VB that the particular data comes from.  In this case, position data
// comes from stream 0.  Normal data comes from stream 1.  Texture coordinate
// data comes from stream 2.
const D3D10_INPUT_ELEMENT_DESC vertlayout_singleindex[] =
{
    { "SV_POSITION", 0, DXGI_FORMAT_R32G32B32_FLOAT, 0, 0, D3D10_INPUT_PER_VERTEX_DATA, 0 },
    { "NORMAL",      0, DXGI_FORMAT_R32G32B32_FLOAT, 1, 0, D3D10_INPUT_PER_VERTEX_DATA, 0 },
    { "TEXCOORD0",   0, DXGI_FORMAT_R32G32_FLOAT,    2, 0, D3D10_INPUT_PER_VERTEX_DATA, 0 },
};
UINT iNumElements = sizeof(vertlayout_singleindex)/sizeof(D3D10_INPUT_ELEMENT_DESC);
D3D10_PASS_DESC PassDesc;
g_pRenderScene_SI->GetPassByIndex( 0 )->GetDesc( &PassDesc );
V_RETURN( pd3dDevice->CreateInputLayout( vertlayout_singleindex, iNumElements, PassDesc.pIAInputSignature, &g_pVertexLayout_SI ) );
```

Before rendering the sample must make sure that all streams are bound, that the index buffer is bound, and that the correct input layout is set. Notice that the third vertex stream (stream number 2) can be switched between the original set of texture coordinates and the alternate set of texture coordinates without affecting the other streams.

```
UINT strides[3];
UINT offsets[3] = {0, 0, 0};

// Set the parameters for MultiIndex or SingleIndex
ID3D10Buffer* pBuffers[3];
```

```
...

    pd3dDevice->IASetInputLayout( g_pVertexLayout_SI );

    pBuffers[0] = g_pVBs[ST_VERTEX_POSITION];
    pBuffers[1] = g_pVBs[ST_VERTEX_NORMAL];

    if( g_bUseAltUV )
        pBuffers[2] = g_pVBs[ST_VERTEX_TEXTUREUV2];
    else
        pBuffers[2] = g_pVBs[ST_VERTEX_TEXTUREUV];

    strides[0] = sizeof(D3DXVECTOR3);
    strides[1] = sizeof(D3DXVECTOR3);
    strides[2] = sizeof(D3DXVECTOR2);

    numVBsSet = 3;

...

pd3dDevice->IASetVertexBuffers( 0, numVBsSet, pBuffers, strides, offsets );
pd3dDevice->IASetIndexBuffer( g_pIB, DXGI_FORMAT_R32_UINT, 0 );
pd3dDevice->IASetPrimitiveTopology( D3D10_PRIMITIVE_TOPOLOGY_TRIANGLELIST );
```

Multi-Stream Multi-Index Rendering

The Direct3D 10 codepath also allows the use of vertex streams that are stored at different frequencies. In order to access this vertex data stored at different frequencies, multiple index buffers are needed. Note that there is no API to allow users of Direct3D 10 to set more than one index buffer at a time. To perform this multi-stream multi-index rendering, some fancy shader work must be done.

First the buffers are created. In this scenario, 5 streams are created. They are described as follows:

- FewVertexPositions: Stores NumVertices unique positions.
- PositionIndices: Stores NumFaces*3 indices into the position buffer.
- VertexTextureUV: Stores NumFaces*3 texture coordinates.
- VertexTextureUV2: Stores NumFaces*3 texture coordinates.
- FaceNormals: Stores NumFaces vertex normals.

Note that FewVertexPosition holds NumVertices positions, while PositionIndices, VertexTextureUV, and VertexTextureUV2 hold NumFaces*3 values. In addition, FaceNormals only contains NumFaces values. This shows a large difference in data frequencies since NumVertices != NumFaces*3 in this scenario.

Since the position indices and texture coordinates share the same frequency, they can be added to the input layout.

```
// Create a Input Layout for the MultiStream MultiIndex data.
//
const D3D10_INPUT_ELEMENT_DESC vertlayout_multiindex[] =
{
    { "POSINDEX",    0, DXGI_FORMAT_R32_UINT,              0, 0, D3D10_INPUT_PER_VERTEX_DATA, 0 },
    { "TEXCOORD0",   0, DXGI_FORMAT_R32G32_FLOAT,    1, 0, D3D10_INPUT_PER_VERTEX_DATA, 0 },
};
iNumElements = sizeof(vertlayout_multiindex)/sizeof(D3D10_INPUT_ELEMENT_DESC);
g_pRenderScene_MI->GetPassByIndex( 0 )->GetDesc( &PassDesc );
V_RETURN( pd3dDevice->CreateInputLayout( vertlayout_multiindex, iNumElements, PassDesc.pIAInputSignature, &g_pVertexLayout_MI ) );
```

They are also bound as vertex buffers similarly to the way it is handled for multi-stream single-index rendering.

```
    pd3dDevice->IASetInputLayout( g_pVertexLayout_MI );

    pBuffers[0] = g_pVBs[ST_POSITION_INDEX];
    if( g_bUseAltUV )
        pBuffers[1] = g_pVBs[ST_VERTEX_TEXTUREUV2];
    else
        pBuffers[1] = g_pVBs[ST_VERTEX_TEXTUREUV];

    strides[0] = sizeof(UINT);
    strides[1] = sizeof(D3DXVECTOR2);
```

However, the data stored at different frequencies must be passed in as buffers.

```
    g_pPosBuffer->SetResource( g_pFewVertexPosRV );
    g_pNormBuffer->SetResource( g_pFaceNormalRV );
```

The rest of the magic happens in the vertex shader. The vertex position is loaded from the position buffer based upon the input position index. The face normal is loaded from the normal buffer using the vertexID/3. SV_VertexID is automatically generated by the input assembler.

```
float4 Pos = g_posBuffer.Load( input.PositionIndex );
float4 Norm = g_normBuffer.Load( input.vertID/3 );
```