Technical Report

Mesh Instancing

Abstract

What is Mesh Instancing?

Before we talk about instancing, let's briefly talk about the way that most D3D applications work. In order to draw a polygonal object on the screen in D3D, you call one of a handful of Draw* functions. These vary in what they do, but they all source a bank of vertex data, and marshal it into polygons that are drawn by a graphics card. When drawing, if any of the polygons differ in applied texture or vertex/pixel constants/shaders, separate draw calls are needed, since an API call needs to be made to change those states. There is a non-zero overhead associated with each draw and state-changing API call, as both D3D and the driver do work to marshal the data sent to the card with each draw call. It is this overhead that helps make a significant portion of 3D applications CPU bound.

NVIDIA's Geforce 6 class hardware supports the DX9 Instancing API, which allows users to draw multiple instances of the same mesh with a single draw call. This feature reduces the CPU overhead of small batch sizes, and helps the CPU not be the bottleneck in your 3D application.

Bryan Dudash
bdudash@nvidia.com

NVIDIA Corporation
2701 San Tomas Expressway
Santa Clara, CA 95050

5/12/2004

How Does It Work?

Instancing makes use of the DX vertex stream frequency API. Microsoft has overloaded this API to support instancing. In this section we give an overview of the way the system used to work, and how it has been updated.

Vertex Stream Frequency Divider API

Before the instancing API, there was simply the vertex stream frequency divider. His allowed the user to have some control over how the data in a vertex buffer was iterated over. It is controlled via the method:

IDirect3dDevice9::SetStreamSourceFreq(UINT StreamNumber, UINT Setting)

This method was used to specify the stream divider for the specified stream. That is, when iterating over the VB, the vertex processing system would only increment the element pointer in the stream once for every N "vertices" processed. This means that with a divider of 2, each element in the stream would be used twice.

Same method, New Usage

This method is the primary interface to the instancing API. New in DX9.0c, Microsoft has reserved the upper two bits of the *Setting* parameter for control keys. This allows the same API to be used to control instancing. The two new control bits are defined below. Simply use bitwise OR to include the control bits at the top of your value to invoke the special logic.

Control Key	Description and Setting Value
`D3DSTREAMSOURCE_INDEXEDDATA`	When this control bit is set, the *Setting* value represents the number of instances to draw. That is, the number of times to loop the stream. This control word can only be on the primary stream.
`D3DSTREAMSOURCE_INSTANCEDATA`	When this control bit is set, the *Setting* value represents a divider over the number of instances. Usually this value should be set to 1.

A Simple Example

Let's look at a simple example. Let's say you have a tree mesh that you want to instance. The mesh is 100 polys, and you want to draw 10,000 of them. Without instancing, you would be forced to reduce the draw count, or come up with a creative solution (making a big VB with pre-transformed trees) because 10,000 draw calls will bring even the beefiest CPU to its knees. Let's see what instancing does with it. Below is some pseudo code:

```
- Create a vertex and index buffer containing a single copy of the tree mesh
- Create a custom vertex buffer containing 3 D3DXVECTOR3's which will represent the
per-instance world transforms
- Create a vertex declaration with 2 streams as mentioned above.
- Foreach(Frame)
        - Possibly update matrix VB to change positions or # of instances
        - Set declaration, and stream data
        - SetStreamSourceFreq(0, D3DSTREAMSOURCE_INDEXEDDATA | numInstances)
        - SetStreamSourceFreq(0, D3DSTREAMSOURCE_INSTANCEDATA | 1))
        - DrawIndexedPrimitive(D3DPT_TRIANGLELIST,0,0,numVerts,0,numIndices/3));
```

In this example, if the tree was 100 polys, then the IB would contain 300 indices. In this example, if we were drawing 10,000 trees, and they are not moving, then we would have everything on the card, and only be issuing a single draw call. Our CPU overhead would be VERY low. Of course, there is some overhead associated with using instancing. We talk about this overhead in the performance section below.

What Is REALLY Going On?

What is really going on with instancing is that the primary stream uses an extension to the vertex stream frequency controls; Modulus. The primary stream has a modulus of its size, such that when vertex processing reaches the end of the buffer, it loops back to the beginning. The secondary streams make use of the divider functionality, however, they divide (under the hood) with *Stream1Size*Divider* which effectively has them increment themselves at a max pace of once per **instance** as opposed to once per vertex.

Take *figure 1* below. It shows to streams. Stream 0 contains the mesh vertices. In this example there are 4 verts. Stream 1 contains 3 per instance world matrices. The implication here is that we only have 3 instances that we wish to draw. When drawing these using the instancing API, all 4 mesh verts are iterated over 3 separate times. However, each instance matrix is iterated over only once. Let's look at a subsection of the processing.

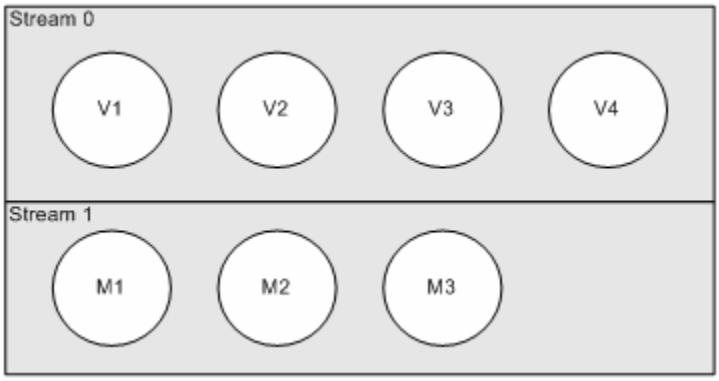

Figure 1: Stream behavior example. Stream 0 contains 4 mesh vertices, and stream 1 contains 4 per instance matrices.

When the GPU begins to draw these instances, it starts with both stream pointers at the beginning. This means we invoke the vertex shader passing down the data from V1&M1. After that vertex is processed, the V2&M1 data is passed down. Followed by V3&M1. After all vertices for the first instance have been processed like this, the pointer for stream 0 gets reset back to the beginning and the pointer for stream 1 gets incremented to point to the next instance matrix. It goes through the streams in this manner for as many instances as you specify via the API calls.

Performance Implications

Use of instancing isn't free. There is a small amount of per instance overhead in the driver (luckily this is much less than normal draw calls). Also, since we are passing down extra instance data in the vertex stream, all the instance data adds to our vertex stride, which will reduce our vertex cache efficiency. As well, the instance data may require that we do extra math ops per-vertex when we could be doing that math per-instance. Since with instancing, you might pass down the world transform, you may have to do a matrix multiply to obtain the WorldViewProjection matrix in the vertex shader. None of these concerns are too bad, and in many situations, instancing is a win.

Below is some empirical performance data. The application draws one million diffuse shaded polys to the screen. The number of polys is fixed as we vary the batch size, and thus the number of instances required to draw one million polys changes. This means that for non-instanced, this increases the # of draw calls.

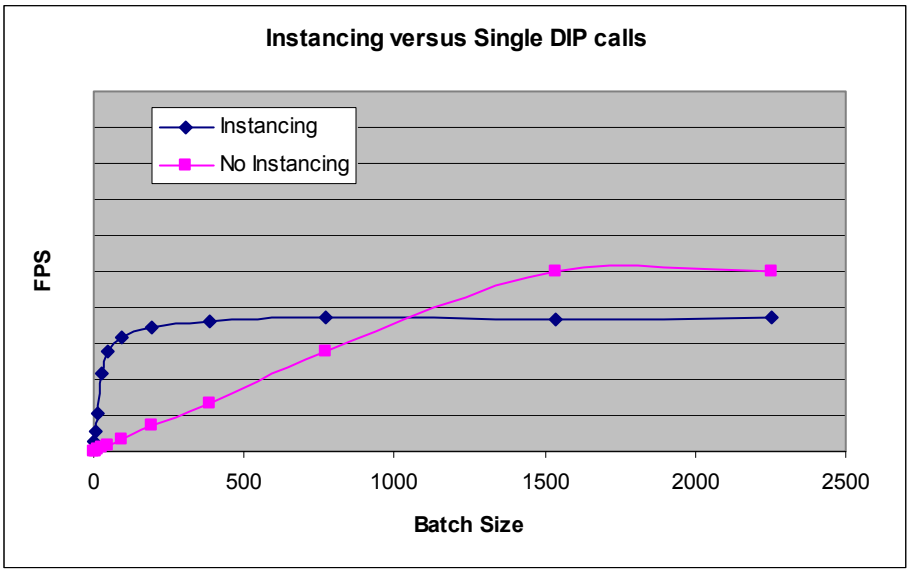

Figure 2: Relative performance of instancing versus non for a fixed million poly scene. Performance varies based on batch size.

When to use instancing

It, of course, depends. The most general statement is that if your application is CPU bound, and you have many copies of the same source mesh placed around your world, then you should use instancing.

Empirical performance testing indicates that there is a "sweet spot" for the use of instancing.

First we assume that your scene poly count is bound to some upper bound by the transform speed of the hardware. Over the general range of possible graphics applications, as the size of an instanced mesh gets larger, the number of draw calls per scene will in general be going down. This means that as the size of the instanced mesh gets larger, the total amount of time spent in the overhead of each draw call goes down. Since instancing has a higher per poly overhead, as the per-poly overhead becomes the limiting factor, the benefit from instancing vanishes. In fact, if your application is not CPU and draw call bound, then you may not see any benefit from instancing at all.

When to NOT use instancing

If your application is heavily GPU bound or you have all unique mesh objects in your world.

Instancing has the benefit of spending less time in DX and in the driver preparing data for a begin/end pair. Instancing reduces this overhead; however, if you have plenty of CPU to spare, then instancing won't help you. Currently, not many applications are GPU bound.

A More Complex Example

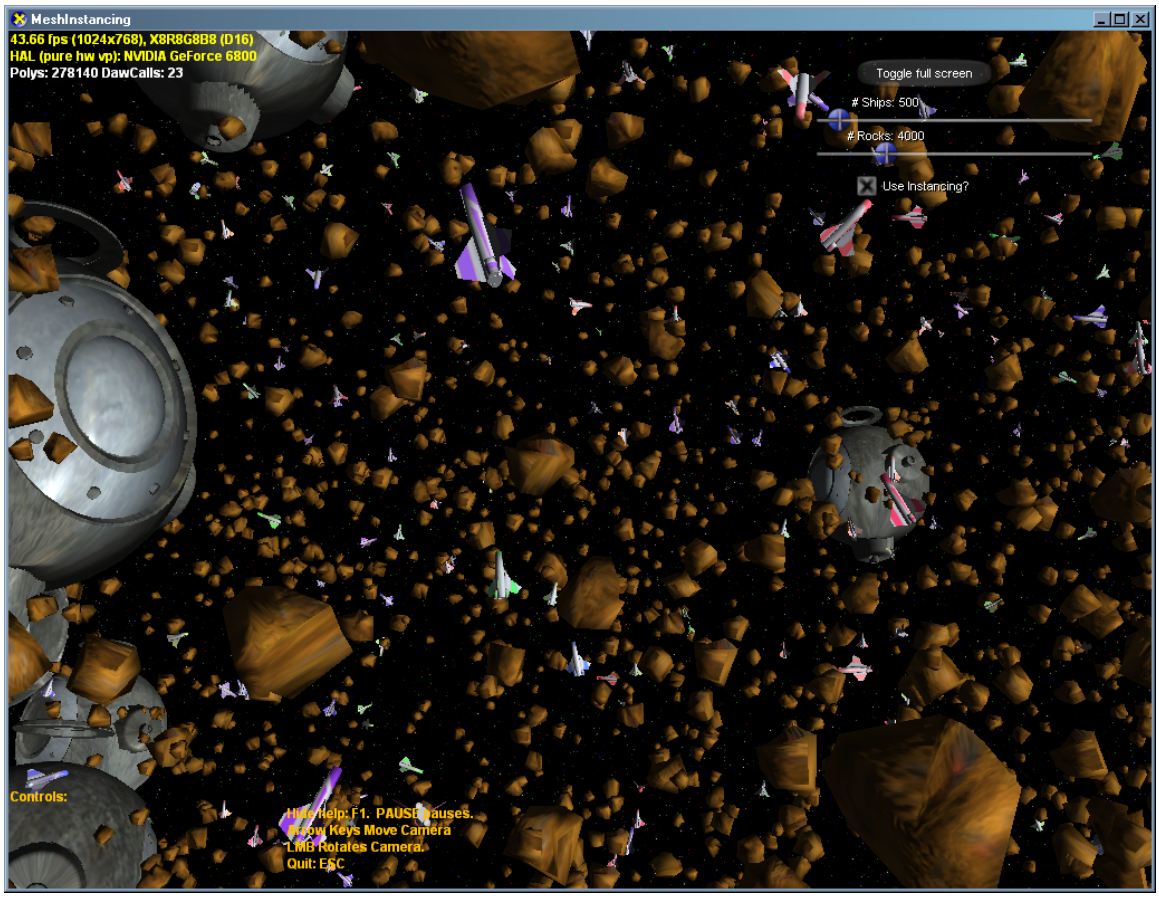

Figure #: Shot of NVSDK Instancing Sample

Now that we've covered all the basics; let's dissect a slightly more complex example. This example was written as a demo for the instancing functionality. As you can see from the screen shot below; it consists of a space scene filled with asteroids and space ships. This sample uses the DX Instancing API as described to reduce the # of draw calls and increase performance.

The Vertex Declaration

There are large mother ships in this sample, but those are added for garnish and are not instanced. There are two meshes being instanced in this demo; the asteroids and the ships. Both use the same vertex definition. This allows the sample to use the same vertex shader to transform all vertices in the system. The vertex definition for both is below. We encode the world transform, and a per instance color in the stream.

```
const D3DVERTEXELEMENT9 g_aMeshVertDecl[] =
{
        {0,  0, D3DDECLTYPE_FLOAT3,   D3DDECLMETHOD_DEFAULT, D3DDECLUSAGE_POSITION, 0},
        {0, 12, D3DDECLTYPE_FLOAT3,   D3DDECLMETHOD_DEFAULT, D3DDECLUSAGE_NORMAL, 0},
        {0, 24, D3DDECLTYPE_FLOAT3,   D3DDECLMETHOD_DEFAULT, D3DDECLUSAGE_TANGENT, 0},
        {0, 36, D3DDECLTYPE_FLOAT3,   D3DDECLMETHOD_DEFAULT, D3DDECLUSAGE_BINORMAL, 0},
        {0, 48, D3DDECLTYPE_FLOAT2,   D3DDECLMETHOD_DEFAULT, D3DDECLUSAGE_TEXCOORD, 0},
        {1, 0,  D3DDECLTYPE_FLOAT4,   D3DDECLMETHOD_DEFAULT, D3DDECLUSAGE_TEXCOORD, 1},
        {1, 16, D3DDECLTYPE_FLOAT4,   D3DDECLMETHOD_DEFAULT, D3DDECLUSAGE_TEXCOORD, 2},
        {1, 32, D3DDECLTYPE_FLOAT4,   D3DDECLMETHOD_DEFAULT, D3DDECLUSAGE_TEXCOORD, 3},
        {1, 48, D3DDECLTYPE_FLOAT4,   D3DDECLMETHOD_DEFAULT, D3DDECLUSAGE_COLOR, 0},
        D3DDECL_END()
};
```

Code sample 1: Vertex definition for instances

The Vertex Shader(s)

All meshes in the system are drawn using MS Effect API (.FX files), and all shaders are written in HLSL and compiled to the appropriate targets. Below is the vertex shader used. Note that with some cleverness in HLSL, you can actually use the same vertex shader for both instancing and non-instancing by providing a wrapped that maps the transforms from either the vertex stream (instanced) or the constants (regular drawing). I am only providing the declaration for the shared vertex shader, as the internals aren't important. It is a basic VS that transforms the vert to its proper projection space position, outputs a tangent to world space basis, passes down a vertex color, etc. The important bits are the wrapper functions that source the relevant data from different places.

```
VS_OUTPUT SharedVS(float4 vPos, float3 vNormal,float3 vTangent,float3 vBinormal,
                           float2 vTexCoord0,float4x4 mWorld,float4 cInstanceColor);

VS_OUTPUT NormalVS(float4 vPos : POSITION,
                           float3 vNormal : NORMAL,
                           float3 vTangent : TANGENT,
                           float3 vBinormal : BINORMAL,
                           float2 vTexCoord0 : TEXCOORD0)
{
   return SharedVS(vPos,vNormal,vTangent,vBinormal,vTexCoord0,g_mWorld,g_MaterialDiffuseColor);
}

VS_OUTPUT InstancedVS( float4 vPos : POSITION,
                           float3 vNormal : NORMAL,
                           float3 vTangent : TANGENT,
                           float3 vBinormal : BINORMAL,
                           float2 vTexCoord0 : TEXCOORD0,
                           float4 vInstanceMatrix1 : TEXCOORD1,
                           float4 vInstanceMatrix2 : TEXCOORD2,
                           float4 vInstanceMatrix3 : TEXCOORD3,
                          float4 cInstanceColor : COLOR)
{
  VS_OUTPUT Output;

  // We've encoded the 4x3 world matrix in a 3x4,
  // so do a quick transpose so we can use it in DX
  float4 row1 = float4(vInstanceMatrix1.x,vInstanceMatrix2.x,vInstanceMatrix3.x,0);
  float4 row2 = float4(vInstanceMatrix1.y,vInstanceMatrix2.y,vInstanceMatrix3.y,0);
  float4 row3 = float4(vInstanceMatrix1.z,vInstanceMatrix2.z,vInstanceMatrix3.z,0);
  float4 row4 = float4(vInstanceMatrix1.w,vInstanceMatrix2.w,vInstanceMatrix3.w,1);
  float4x4 mInstanceMatrix = float4x4(row1,row2,row3,row4);
  return SharedVS(vPos,vNormal,vTangent,vBinormal,vTexCoord0,mInstanceMatrix,cInstanceColor);
}
```

Code Sample 2: vertex shaders using instancing and not using instancing.

Other Miscellaneous

The scene is lit with separate pixel shaders for each object. They all do the same Blinn diffuse, specular, bump-mapped lighting. The exception is the space ships. They have a colorization channel defined in the z component of the normal map. This is used in conjunction with the passed down instance color to colorize the paint on the space ships, while leaving the rest of the ship sourced from the ship texture. This allows the space ships to all be "silver-white metal", but have custom, random colored stripe decals on them.

The Results

This sample shows the use of instancing, in a slightly more realistic setting. The space ships and asteroids are being simulated on the CPU. There is a simple collision system and physics system in place. The scene consists of 4000 asteroids, independently rotating, and 500 space ships, which are attempting to fly around the obstacles. There are also 10 large mother-ships being drawn with normal separate draw calls.

Under these conditions, on a P4 3Ghz machine and an GeForce6800, the scene renders between 35-45fps when instancing, and between 8-11fps when not using instancing. The difference is quite noticeable.

Also, worth reminding that the pixel shaders being used are identical, as is the rendering output.