Skinned Instancing

Bryan Dudash
bdudash@nvidia.com

14 February 2007

Document Change History

Version	Date	Responsible	Reason for Change
1.0	2/14/07	Bryan Dudash	Initial release
2.0	7/26/07	Bryan Dudash	Update to new sample code and technique

Abstract

With game rendering becoming more complex, both visually and computationally, it is important to make efficient use of GPU hardware. Instancing allows you to potentially reduce CPU overhead by reducing the number of draw calls, state changes, and buffer updates. This technique shows how to use DX10 instancing, and vertex texture fetches to implement instanced hardware palette-skinned characters. The sample also makes use of constant buffers, and the SV_InstanceID system variable to efficiently implement the technique. With this technique we are able to realize almost ~10,000 characters, independently animating with different animations and differing meshes at 30fps on an Intel Core 2 Duo GeForce 8800GTX system (see figure 1a and 1b).

Figure 1a. That's a lot of animating dwarves!

Motivation

Our goal with this technique is to efficiently use DirectX 10 to enable large scale rendering of animated characters. This technique can be used for crowds, audiences, etc. and is generally applicable for any situation where there is a need to draw a large number of actors, each with a different animation, and different mesh variations. Inherent in these situations is the idea that some characters will be closer than others, and thus and LOD system is important. The technique can enable game designers to realize large dynamic situations previously not possible without pre-rendering, or severely limited uniqueness of the characters in the secne.

Figure 1b. Close-up shot. Detail changes as camera moves.

How Does It Work?

Traditionally instancing has not been able to render animated objects efficiently. With the advent of DirectX10, efficient skinned animated instancing become possible.

"Skinned Instancing" is an extension of regular instancing. It renders each character using hardware palette skinning. Instead of the standard method of storing the animation frame in shader constants, we encode all frames of all animations into a texture and lookup the bone matrices from that texture in the vertex shader. Thus we can have more than one animation, and each character can be in a different frame of each animation.

We encode the per-instance parameters into a constant buffer and index into that array using the SV_InstanceID.

To achieve mesh variation per instance we break the character into sub-meshes which are individually instanced. This would be meshes such as different heads, etc.

Finally, to avoid work for characters in the far distance we implement an LOD system with lower poly mesh subsections. The decision of which LOD to use is calculated per frame on a per instance basis.

A simple rendering flow is below. For details, please see the subsections.

CPU

- Perform game logic(animation time, AI, etc)
- Determine a LOD group for each instance and populate LOD lists.
- For each LOD
  - For each sub-mesh
    - Populate instance data buffers for each instanced draw call
    - For each buffer
      - DrawInstanced the sub-mesh
- **GPU**
- Vertex Shader
  - Load per-instance data from constants using SV_InstanceID
  - Load bone matrix of appropriate animation and frame
  - Perform palette skinning
- Pixel Shader
  - Apply per-instance coloration as passed down from vertex shader
  - (optional) Read a custom texture per instance from a texture array.

Constants Based Instancing

The technique does not use the traditional method of encoding the per-instance data into a separate vertex stream. In testing we found that reading per-instance data from constant buffer using an index per instance was faster due to better cache utilization of the vertices. With the traditional method, vertex attributes increase by the # of per-instance data attributes. We chose instead to put the low frequency data into constant memory.

SV_InstanceID

Under DirectX10 there are a number of useful variables that can be automatically generated by the GPU and passed into shader code. These variables are called "system variables". All these variables have a semantic that begins with "SV_". SV_InstanceID is a GPU generated value available to all vertex shaders. By binding a shader input to this semantic, the variable will get an integral value corresponding to the current instance. The first index will get 0 and subsequent instances will monotonically increase this value. Thus every instance through the render pipeline gets a unique value and every vertex for a particular instance shares a common SV_InstanceID value.

This automatic system value allows us to store an array of instance information in a constant buffer and use the ID to index into that array. Since we are injecting per-instance data into constant buffers, we are limited in the number of instances we can render per draw call by the size of the constant memory. In DirectX10 there is a limit of 4096 float4 vectors per constant buffer. The number of instances you can draw with this size depends on the size of the per-instance data structure. In this sample we have the following per instance data:

Listing 1. Instance Data structure and constant buffer

```
struct PerInstanceData
{
    float4 world1;
    float4 world2;
    float4 world3;
    float4 color;
    uint4    animationData;
};

cbuffer cInstanceData
{
    PerInstanceData    g_Instances[MAX_INSTANCE_CONSTANTS];
}
```

As you can see, in this sample each instance takes up 5 float4 vectors of constant memory, and so that means we can store a max of 819 instances. So we split each group of instanced meshes into N buffers where N = Total Instances / 819. This is a very acceptable number, and means that if we were to draw 10,000 meshes it would take 13 draw calls. There is a difference in CPU overhead between 1 and 13 draw calls per frame, but the difference between 13 and 819 is much larger. Each draw call removed allows a reduction in CPU overhead, and possible performance. Thus, there is often little effective difference in final framerate between 1 and 13 draw calls.

On the CPU side, the data looks like the following:

Listing 2. C++ Side Instance data struct.

```
struct InstanceDataElement
{
    D3DXVECTOR4 world1;
    D3DXVECTOR4 world2;
    D3DXVECTOR4 world3;
    D3DXCOLOR color;

    // offset in vectors(texels) into the whole data stream
    //  for the start of the animation playing
    UINT animationIndex;
    // offset in vectors(texels) into the animation stream
    //  for the start of the frame playing
    UINT frameOffset;
    UINT unused1;  // pad
    UINT unused2;  // pad
};
```

Palette Skinning using an Animation Texture

In traditional matrix palette skinning, you encode the transform matrices into vertex shader constants. In our case, each character has a different pose, and possible a different animation. We use a texture to store the animation data since the amount of data required for all animation frames is too large to fit into constant memory. DirectX10 class hardware should all have sufficient vertex texture fetch capability to make this a fast operation.

We save each bone matrix for each frame for each animation linearly into a texture, and thus can read it out in the vertex shader.

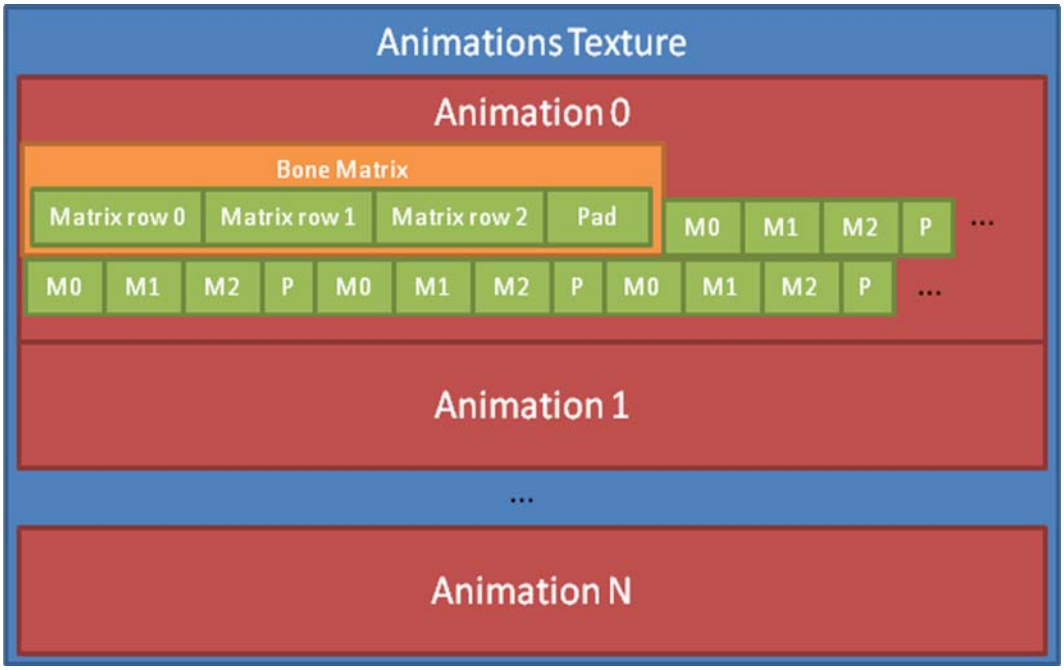

Figure 2. Animations Texture breakdown.

Decode Matrices from a Texture

Below is the function in HLSL that loads the matrix from the animations texture based on the animation offset and bone offset in texels. You can see that we divide and modulus to determine a UV from a linear offset into the animation. Then we point sample 3 rows of the matrix and construct a float4x4 (the translation is encoded in w components or each row). The sampling from the animation texture uses the HLSL Load() method, which returns the texel unfiltered, and takes an exact pixel position as input, thus simplifying the lookup process.

Listing 3. loadBoneMatrix HLSL function

```
// Read a matrix(3 texture reads) from a texture containing animation data
float4x4 loadBoneMatrix(uint3 animationData,float bone)
{
    float4x4 rval = g_Identity;

    // if this texture were 1D, what would be the offset?
    uint baseIndex = animationData.x + animationData.y;

    // 4*bone is since each bone is 4 texels to form a float4x4
    baseIndex += (4 * bone);

    // Now turn that into 2D coords
    uint baseU = baseIndex % g_InstanceMatricesWidth;
    uint baseV = baseIndex / g_InstanceMatricesWidth;

    // Note that we assume the width of the texture
    // is an even multiple of the # of texels per bone,
    // otherwise we'd have to recalculate the V component per lookup.
    float4 mat1 = g_txAnimations.Load( uint3(baseU,baseV,0));
    float4 mat2 = g_txAnimations.Load( uint3(baseU+1,baseV,0));
    float4 mat3 = g_txAnimations.Load( uint3(baseU+2,baseV,0));

    // only load 3 of the 4 values, and deocde the matrix from them.
    rval = decodeMatrix(float3x4(mat1,mat2,mat3));

    return rval;
}
```

Conditional Branching for Weights

As with regular palette skinning, vertices can have up to 4 bone weights, but not all vertices have 4 bones active. Thus we make use of conditional branching in the vertex shader to avoid unnecessary texture loads of matrices we won't use.

Listing 4. Vertex Shader conditional branching

```
VS_to_PS CharacterAnimatedInstancedVS( A_to_VS input )
{
    VS_to_PS output;

    uint4 animationData = g_Instances[input.InstanceId].animationData;

    // Our per instance data is stored in constants
    float4 worldMatrix1 = g_Instances[input.InstanceId].world1;
    float4 worldMatrix2 = g_Instances[input.InstanceId].world2;
    float4 worldMatrix3 = g_Instances[input.InstanceId].world3;
    float4 instanceColor = g_Instances[input.InstanceId].color;

    float4x4 finalMatrix;
    // Load the first and most influential bone weight
    finalMatrix = input.vWeights.x *
                      loadBoneMatrix(animationData,input.vBones.x);

    // Conditionally apply subsequent bone matrices if the weight is > 0
    if(input.vWeights.y > 0)
    {
        finalMatrix += input.vWeights.y *
                          loadBoneMatrix(animationData,input.vBones.y);
        if(input.vWeights.z > 0)
        {
            finalMatrix += input.vWeights.z *
                              loadBoneMatrix(animationData,input.vBones.z);
            if(input.vWeights.w > 0)
                finalMatrix += input.vWeights.w *
                                  loadBoneMatrix(animationData,input.vBones.w);
        }
    }
```

A Few Important Points:

- The animation texture must be a multiple of 4. This allows us to calculate the row only once in the vertex shader and then simply offset along U to get the 4 lines or each matrix.
- We actually encode the matrix into 3 lines to save a texture fetch, but to avoid issues with non power of 2 textures, we add in a pad texel.
- The linear offset in texels to the active animation, and the linear offset within the animation for the active frame are specified per instance.
- The bone index and bone weight information are stored (as normal) per vertex for matrix palette skinning

Geometry Variations

If all characters rendered had the exact same mesh geometry, the user would immediately notice the homogeneousness of the scene and her disbelief would not be suspended. In order to achieve more variation in character meshes, we break a character into multiple pieces and provide alternate meshes. In the case of this sample we have warriors with differing armor pieces, and weapons. The character mesh is broken up into these separate pieces, and each piece is instanced separately.

The basic method for this is to understand which pieces each character instance contains. Then, we can create a list of characters that use a given piece. At draw time, we simply iterate over the pieces, inject proper position information into the per-instance constant buffer, and draw the appropriate amount of instances.

Figure 3. Shot of mesh showing all mesh variations.

Figure 4 is where we can see that the source mesh contains all the mesh permutations for each character. Notice that all the weapons are included in the same file. They are exported as separate mesh objects but all bound to an identical skeleton. This allows us to reuse the animations from the character for all the sub-sections. On load time, the system will generate random permutations of the sub-sections to give each character a different appearance. The technique supports as many mesh variations as your artists can come up with. For a real game, you might want to have finer artist control over this (as opposed to random generation) which would require some work in tools or the export path.

LOD System

Because characters in the distance take up fewer pixels on the screen, there is no need for them to be as high poly as characters closer to the camera. In addition, distant characters do not need to sample from the normal map, or calculate complex lighting. Thus we implement an LOD system to improve performance. The technique for instancing breaks each character into a collection of mesh pieces that are instanced. An LOD system is easily implemented by simply adding more pieces to the instancing system. Every frame, each character instance determines the LOD group that it is in by its distance from the camera. This operation happens on the CPU. Then at render time, collections of each mesh piece in each LOD group are drawn. As we iterate through each LOD mesh piece, we consult which instances are in that LOD group and use that piece. Thus we can update the instance data buffers appropriately to render each character at the correct LOD level. We also perform a simple view frustum culling on the CPU to avoid sending thousands of characters behind the camera to the GPU.

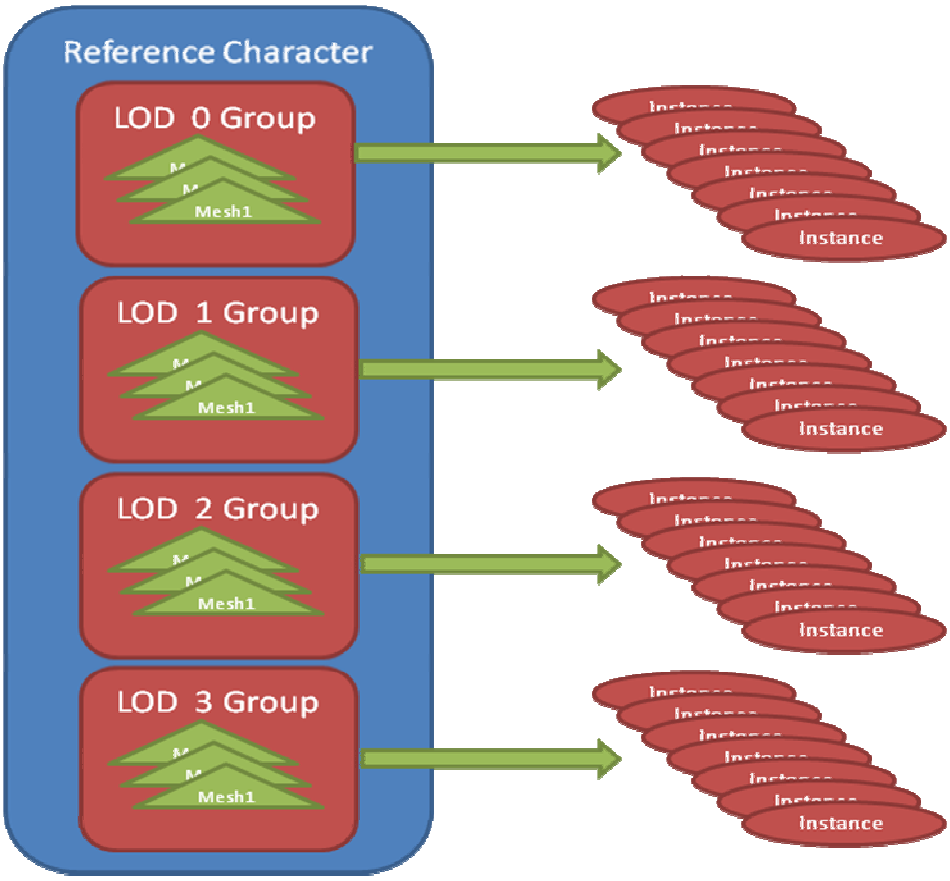

Figure 4. LOD Data layout

An additional benefit of an LOD system is to reduce geometric aliasing, which would result if you were to draw a high poly object in the far distance such that vertices overlapped on a pixel.

Implementation Details

The source code is divided into a few folders and `cpp` files:

- ***Character*** – contains basic classes for managing CPU side instance data. This is sample specific, and not the most interesting.
- ***Materials*** – contains a class to act as a repository for textures
- ***MeshLoader*** – contains the black heart of evil. The mesh loader classes are not pretty, formatted, or recommended in any way. Nothing to see here.
- ***`SkinnedInstancing.cpp`*** - contains all the basic framework wrapper code to setup the device, etc. This is better explained by the basic tutorials in Microsoft'
- ***`ArmyManager.cpp`*** - contains almost all of the interesting D3D10 code. It creates and maintains all relevant D3D resources, and also has the render code for both instancing and non-instancing cases.
- ***SkinnedInstancing.fx*** – contains all the shader code used in the sample. It has the matrix palette skinning as well as all the shader side instancing support.

Sample Implementation Caveats

There are a number of things that this sample does that are sub-optimal, or something that you would never do in a real game title. I list these below along with some explanation of why they were implemented in this way.

File loading classes are a mess. This was mostly due to the fact there is no robust animation support in D3DX for DirectX10 yet. The loader classes use a DirectX9 device to load animations and mesh data from an .X file, and then create DirectX10 buffers from that data. This is really a dirty bit of code, and in a real game engine, you would have a established data loading path, and thus wouldn't have to worry about how to get access to the mesh and animation data. This section of the sample should be avoided.

Mesh assignment to different attachment groups is based on a text parsing of the mesh name. If the mesh was exported with an "attachment_" name, then it gets flagged as an attachment and the bit flag for it is registered. In a real game engine scenario, you might want to consider a more robust, and possibly artist controlled system.

Running the Sample

REF will be unusably slow for large numbers of instances.

There are no other special considerations when running the sample. There are some keyboard shortcuts.

ESC – Exit

F1 – Display help

L – show lighting GUI to tweak constants.

Performance

As with any type of instancing, performance gains are seen on the CPU side. By using instancing, you free up some CPU processing time for other operations, such as AI, or physics. Thus performance of this technique depends on the CPU load of your game.

Note: In general, any instancing technique will shift the load from the CPU to the GPU, which is a good thing, since the CPU can always do more processing of your game data.

Performance also depends on where you set the LOD levels. If all the characters are rendered at the highest LOD, then you can render much less characters. But as you bring the lines for far LODs closer to the camera, you may see artifacts. This is a judgment call of the graphics programmer or designers.

Note: This is running on an Intel Core 2 2.93Ghz system and a GeForce 8800GTX.

You can gain more performance with more aggressive use of LOD, and you can gain more quality with less aggressive use of LOD.

Integration

Integration is more like integration of a new rendering type. Most likely you would define a crowd, or background group of animated characters. The placement of the characters in the group can be specified by artists, and used to populate the instance data buffer. The mesh data and animation data is exactly the same as you would expect for a normal hardware palette skinning implementation. The only difference is that you need to preprocess the animation curves into a collection of bone matrices per frame. This should most likely be a preprocessing step.

References

Carucci, Francesco. 2005. "Inside Geometry Instancing" In GPU Gems 2, edited by Randima Fernando, pp XX-XX. Addison-Wesley Professional

Dudash, Bryan. 2005 "Technical Report: Instancing" In NVIDIA SDK 9.5, http://developer.nvidia.com

Microsoft. 2007. "DirectX Documentation for C++" In Microsoft DirectX SDK (February 2007). http://msdn.microsoft.com/directx

Dudash, Bryan. 2007. "Skinned Instancing" In NVIDIA SDK10, http://developer.nvidia.com

NVIDIA Corporation
2701 San Tomas Expressway
Santa Clara, CA 95050
www.nvidia.com