



# 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

# 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 variables to efficiently implement the technique. The rendered scene shows thousands of individual characters, each with differing meshes, animating in different poses; drawn using just a handful of draw calls.



Figure 1. Screen shot of SkinnedInstancing in action.

# Motivation

Our goal with Skinned Instancing is to efficiently use DirectX 10 to enable large scale rendering of animated characters. This technique can be used for crowds, audiences, etc. The technique is sufficient for any situation where there is a need to draw a large number of characters, each with a different animation, and different mesh variations.

# How Does It Work?

Skinned Instancing is an extension of regular instancing. It renders each character using hardware palette skinning. However, 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 an array of shader constants 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

## Our New Friend SV\_InstanceID

This whitepaper will not go into the basics of instancing [see references], but it will address a new feature in DirectX10 instancing; the system variable SV\_InstanceID. This system variable is automatically generated by the hardware and passed into the vertex shader (if you include it in the input structure). You then have access to it. It is an integer monotonically increasing value that is reset every draw call. Every instance through the render pipeline gets a unique value, and so every vertex for a particular instance shares a common SV\_InstanceID value. This automatic system value did not exist in DirectX9 and is very useful, since it allows us to store an array of all instance information in a constant buffer and use the ID to index into that array. In DirectX9 the user was forced to inject all per instance data into a separate vertex stream; this was cumbersome to use and reduced vertex cache performance.

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:

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

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  $N = \text{TotalInstances} / 819$ .

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

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

    UINT animationIndex;
    UINT frameOffset;
    UINT attachmentSet;
    UINT lerpValue;
};
```

## Geometry Variation

If all characters rendered had the exact same mesh geometry, the user would immediately notice the homogeneity 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.

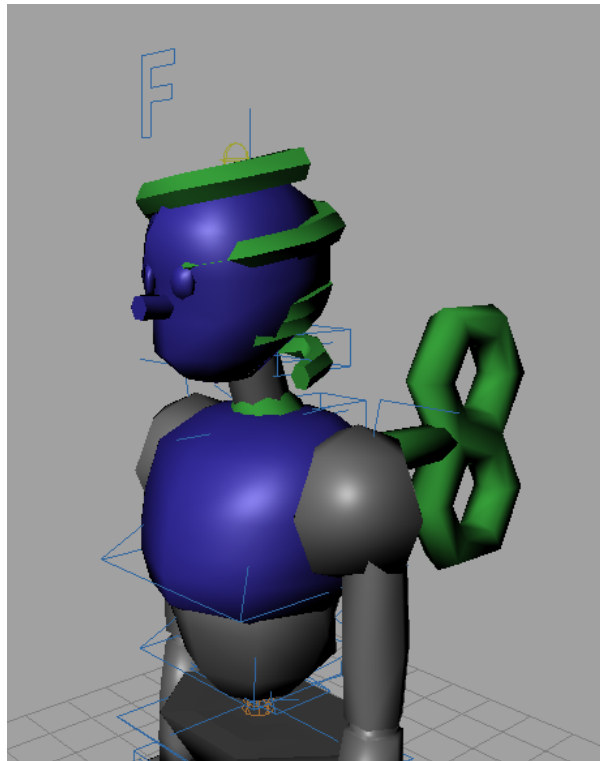


Figure 2 mesh used in this actual sample.

The shot is from Maya where we can see that the source mesh contains all the mesh permutations for each character. They are exported as separate meshes 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.

## Palette Skinning with an Animation Texture

This sample also uses matrix palette skinning to animate all the characters. In traditional matrix palette skinning, you encode the transform matrices into vertex shader constants. In our case, each character has a different pose, and possibly 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.

**Note:** In DirectX10 texture access is orthogonal to the stage of the shader pipeline. Thus fetching a texture in the vertex shader is just as fast as fetching in a pixel shader.

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.

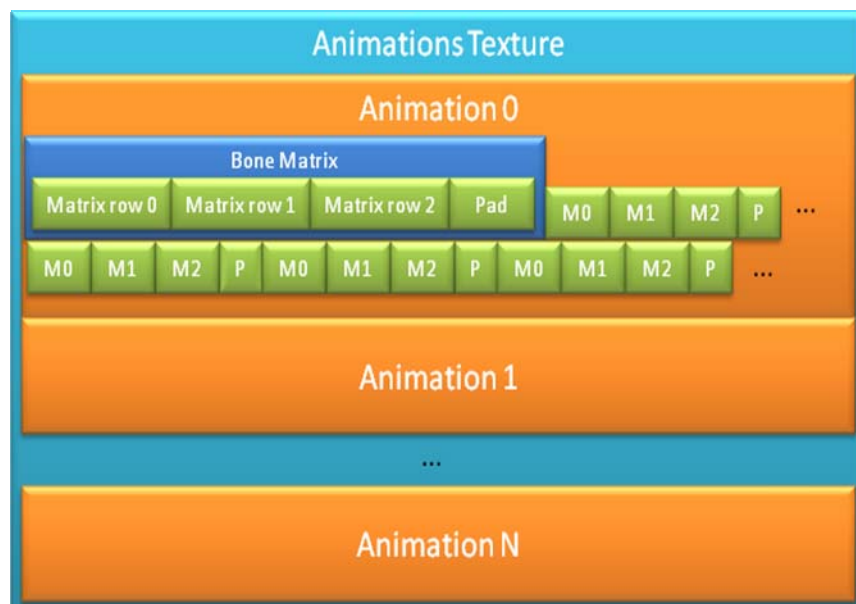


Figure 4: Animations Texture breakdown.

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

Below is the function in the shader that loads the matrix from the animations texture based on the animation offset and bone offset in texel. You can see that we divide and modulus to determine a UV from a linear offset into the

```
// Read a matrix(3 texture reads) from a texture containing animation data
float4x4 loadBoneMatrix(uint3 animationData,float bone)
{
    // Calculate a UV for the bone for this vertex
    float2 uv = float2(0,0);
    float4x4 rval = g_Identity;

    // if this texture were 1D, what would be the offset?
    uint baseIndex = animationData.x + animationData.y;
    baseIndex += (4*bone);    // 4*bone is since each bone is 4 texels to form a float4x4

    // Now turn that into 2D coords
    uint baseU = baseIndex%g_InstanceMatricesWidth;
    uint baseV = baseIndex/g_InstanceMatricesWidth;
    uv.x = (float)baseU / (float)g_InstanceMatricesWidth;
    uv.y = (float)baseV / (float)g_InstanceMatricesHeight;

    // 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
    float2 uvOffset = float2(1.0/(float)g_InstanceMatricesWidth,0);

    float4 mat1 = g_txAnimations.SampleLevel( g_samPoint,float3(uv.xy,0),0);
    float4 mat2 = g_txAnimations.SampleLevel( g_samPoint,float3(uv.xy + uvOffset.xy,0),0);
    float4 mat3 = g_txAnimations.SampleLevel( g_samPoint,float3(uv.xy + 2*uvOffset.xy,0),0);

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

    return rval;
}
```



## 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. Thus we implement an LOD system to improve performance. Since the technique for instancing breaks each character into a collection of mesh pieces that are instanced. An LOD system is simply adding more pieces to the instancing system. Thus every frame, each character instance determines the LOD group that it is in by its distance from the camera. This operation happens on the GPU. 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.

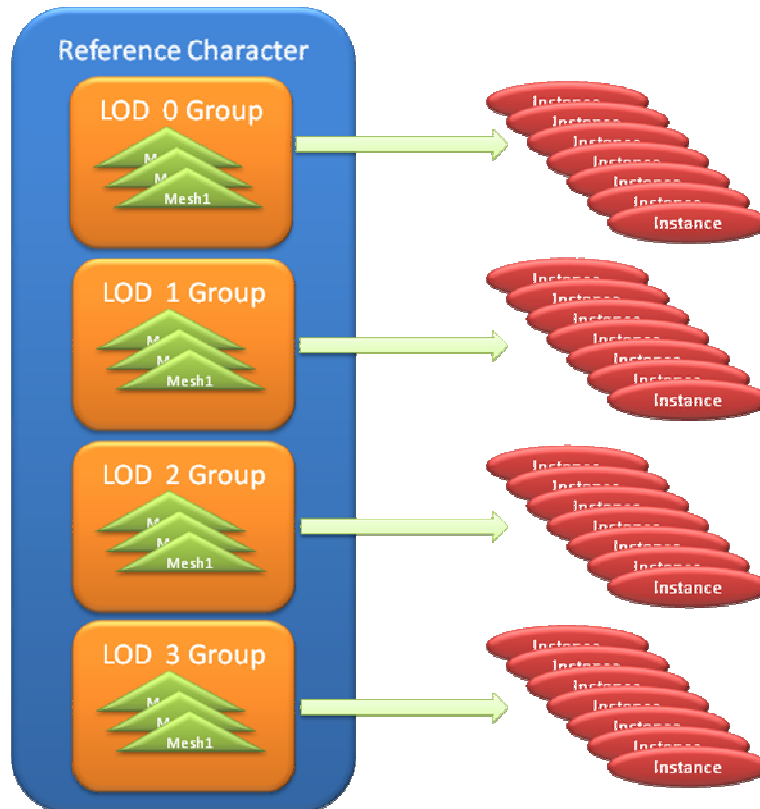


Figure 5: LOD Data layout

# 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<sup>9</sup>
- ***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

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

ESC – Exit

i/I – toggle instancing

k/K – decrease instances

l/L – increase instances

## 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.

This sample in the instanced case runs at about 30fps for 8200 characters. This is with the LOD level lines set to 10, 30, and 50 character radiuses respectively. This results in 47 instanced draw calls. Compared to 24601 single draw calls if you were to draw the same character meshes in a more traditional way.

**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

[Dudash 2005] “Instancing”; NVIDIA SDK 9.5;<http://developer.nvidia.com>

[Francesco Carucci 2005] “Inside Geometry Instancing”; GPU Gems 2  
Programming Techniques for High Performance Graphics and General Purpose  
Computation;Addison-Wesley Professional;ISBN-10:0321335597

[Microsoft 2007] Microsoft DirectX SDK (February 2007) DirectX Documentation  
for C++

**Notice**

ALL NVIDIA DESIGN SPECIFICATIONS, REFERENCE BOARDS, FILES, DRAWINGS, DIAGNOSTICS, LISTS, AND OTHER DOCUMENTS (TOGETHER AND SEPARATELY, "MATERIALS") ARE BEING PROVIDED "AS IS." NVIDIA MAKES NO WARRANTIES, EXPRESSED, IMPLIED, STATUTORY, OR OTHERWISE WITH RESPECT TO THE MATERIALS, AND EXPRESSLY DISCLAIMS ALL IMPLIED WARRANTIES OF NONINFRINGEMENT, MERCHANTABILITY, AND FITNESS FOR A PARTICULAR PURPOSE.

Information furnished is believed to be accurate and reliable. However, NVIDIA Corporation assumes no responsibility for the consequences of use of such information or for any infringement of patents or other rights of third parties that may result from its use. No license is granted by implication or otherwise under any patent or patent rights of NVIDIA Corporation. Specifications mentioned in this publication are subject to change without notice. This publication supersedes and replaces all information previously supplied. NVIDIA Corporation products are not authorized for use as critical components in life support devices or systems without express written approval of NVIDIA Corporation.

**Trademarks**

NVIDIA, the NVIDIA logo, GeForce, and NVIDIA Quadro are trademarks or registered trademarks of NVIDIA Corporation in the United States and other countries. Other company and product names may be trademarks of the respective companies with which they are associated.

**Copyright**

© 2007 NVIDIA Corporation. All rights reserved.

**nvidia.**

NVIDIA Corporation  
2701 San Tomas Expressway  
Santa Clara, CA 95050  
[www.nvidia.com](http://www.nvidia.com)