GPU Blend Shapes

Tristan Lorach
tlorach@nvidia.com

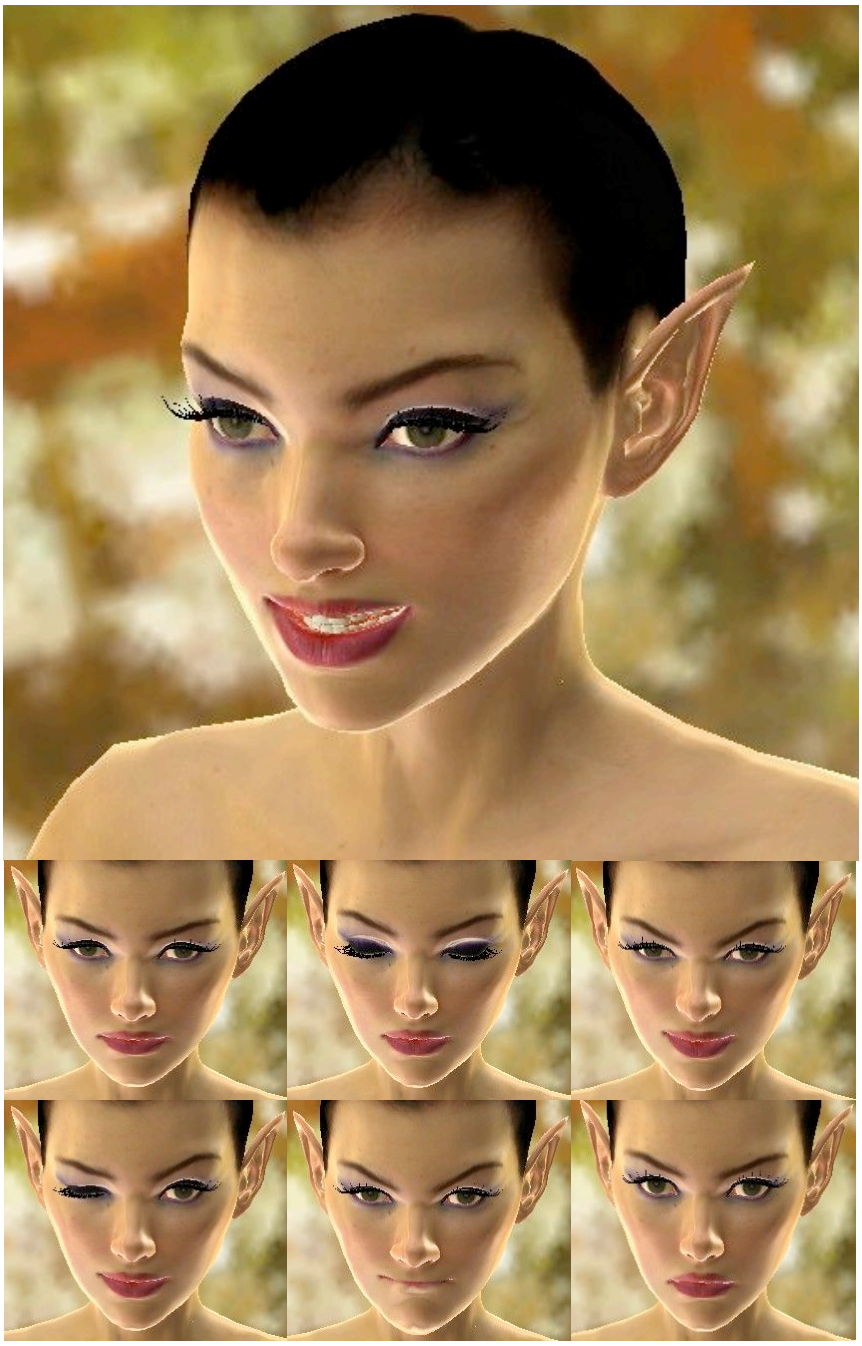
February 2007

Document Change History

Version	Date	Responsible	Reason for Change
1.0	02/16/07	T.Lorach	Initial release

Abstract

This paper presents two methods for computing *Blend Shapes* on the GPU. The technique produces hybrid, blended meshes by combining a large number of input meshes. (The input meshes are also called blend shapes.)



Motivation

Blend Shapes is a well known, useful deformer available in Maya® or any other DCC application. This deformer can achieve simple animations by morphing the mesh from one combination of deformations to another.

The technique can be used for various purposes like facial animation; customization of game characters; full character animation with or without the combination of bones and skinning deformation, etc.

There are many ways of using *Blend Shapes*. Some animations may for example use a sequence of poses through which we will move to get the animated sequence. Other animations may combine several blend shapes together to get a final new expression.

GPU generations prior to GeForce 8800 had to use the vertex attributes to send additional data for blend shapes. Because of the limited amount of available attributes, we were unable to deal with a high number of blend shapes simultaneously. The mix of complex blend shapes could therefore only be done on the CPU. If we wanted to dedicate this job to the GPU we had to dramatically limit the number of blend shapes being used together.

Our work uses NVIDIA's Dawn character. Let's examine how the blend shapes were processed in 2002 by [Beeson04] and his Demo Team, who obviously did not have access to a GeForce 8800.

The NVIDIA Demo Team managed to pack 4 facial expressions in the vertex attributes. So they could create a predefined sequence of facial expressions by slightly moving from one expression (A) to another (B). Then when the animation fully reached expression B, they could move B to A's slot and add a new expression C where B was. The animation could then flow from B to C without a discontinuity. On top of this sequence, they used the two other available slots to add some orthogonal facial expressions, such as eye blinking or lip pouting.

This paper and the related demo show how to turn this blend shape-based deformation into a fully orthogonal effect with no limitations on the number of expressions. The effect is of course fully handled by the GPU.

Our new implementation has three interesting consequences:

1. Mixing a large number of expressions can lead to new expressions. The efficiency of the GPU makes this mix acceptable for real time applications
2. Even if we don't need to mix so many blend shapes together, this method allows us to perform a smooth transition from one combination of expressions to another. For example, we can easily make a transition from a five 'Blend Shape' expression to another one made of seven blend shapes. This technique will allow the GPU to handle the total twelve blend shapes, during the transition.
3. Even if a high amount of blend shapes are being exposed to the shader and available at any time, we will only send the non null expressions (weights>0) to the GPU.

We present two different implementations of this effect:

- one using the 'Stream Out' buffer as many times as needed;
- one looping in the vertex shader, doing all animation in a single draw call.

How Does It Work?

The five major DX10 features we used are:

- Stream Out to vertex buffer (in the first method);
- Slots (formerly called 'Streams' in DX9): the ability of DX10 (and DX9) to mix together different pieces of geometry to feed the GPU with all the needed attributes;
- the vertex shader loop capability;
- Buffer<> template to access generic buffers that we can bind as 'Resource Views';
- the VertexID system value.

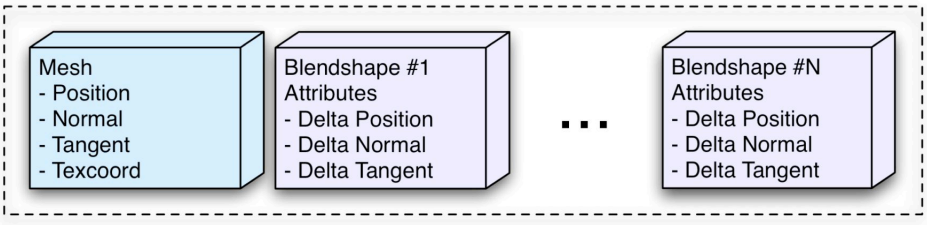
Mesh Definition

Some elements of the scene don't have anything related to our effect : the back of the head and the shoulders won't go through the blend shape computation. Therefore these meshes will just be sent to the GPU as usual.

Then we will compute the expression of Dawn's face by providing to the GPU:

- One neutral mesh: this mesh contains the vertex positions, normals, tangents, and texture UVs.
- 54 expressions (blend shapes): these shapes contain the differences that we need to add to the neutral mesh to reach the new expression. We typically store the positions, normals, and tangents (the binormal vector can be computed). There is no need to store the attributes that do not change (texture coordinates for example).

We used Maya to create the 'Blend Shape' data and we exported the final mesh in a custom format made of a sequence of blocks:

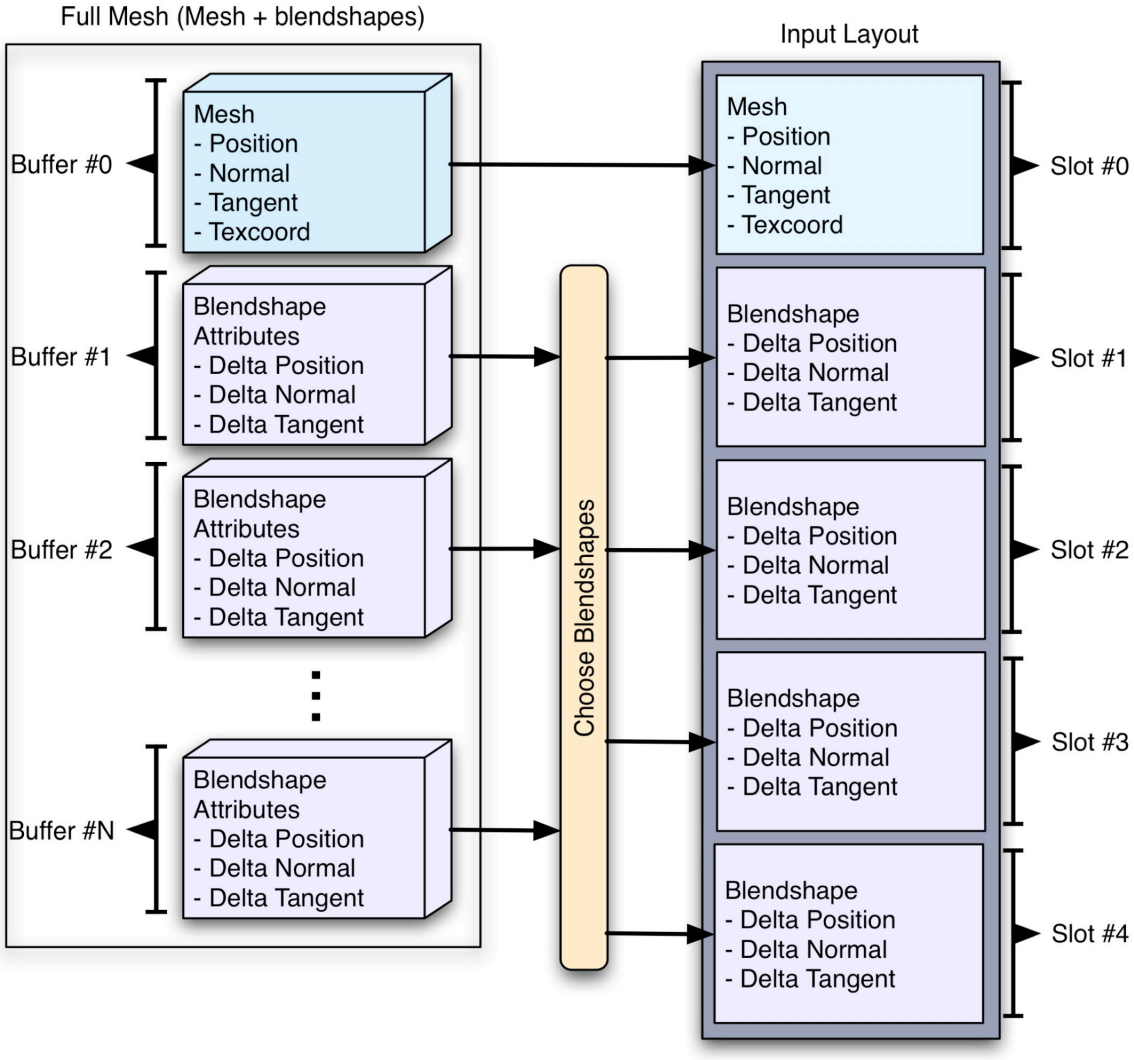


Stream Out Method

In this technique, the vertex buffer of two blend shapes are added to the vertex buffer of the main shape at the time we send the vertices to the GPU.

We use the DirectX10's ability to combine Slots together to provide data to all the vertex attributes:

- The first slot will contain the mesh attributes.
- Two other slots will be added to provide two facial expressions. For efficiency reasons, we try to avoid sending null expressions.

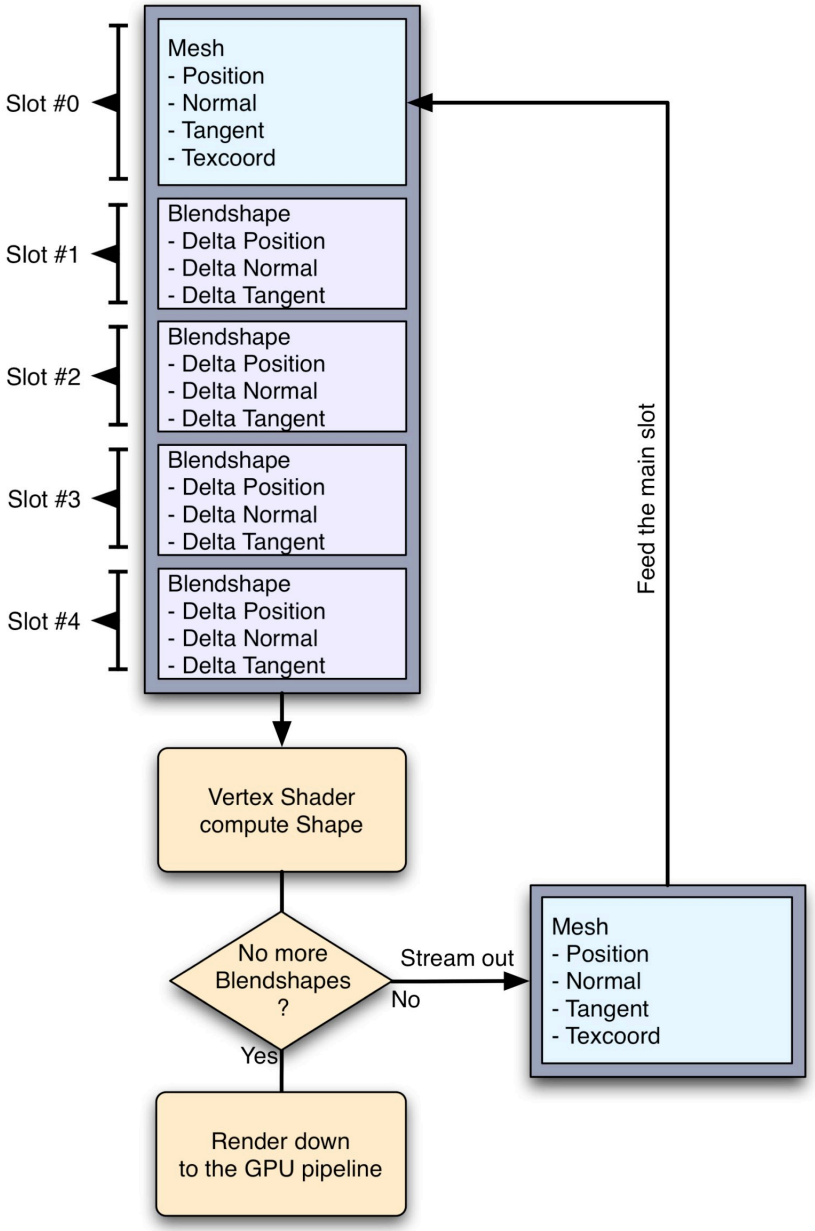


The number of blend shapes we can append depends on how they fit in the 16 available attributes. It could be possible to add more blend shapes in additional slots if we compressed the data (normalized vectors packed in 32bits or using fewer

components…) or if we didn't need to provide 'tangent' vectors. In our case, we arbitrarily decided to use four BlendShapes on top of the main mesh. We leave it as an exercise to the reader to address this limitation.

We use the stream out buffer to loop as many times as needed, accumulating deltas, to end up with the final expression. Note that we never perform any read-back of our data: everything is kept within Vertex/Stream Buffers.

First, we start with the original mesh in Slot #0. Then the next iterations will replace this original mesh with the new one that the GPU streamed out. Our algorithm is using two Buffers and is doing a 'ping-pong' swap : while one is used as the Stream output, the other will be used as the Vertex Buffer for Slot #0. The process will end when no more active BlendShapes need to be added.



Buffer Template Method

DirectX10 allows a shader to access data in a buffer that we can bind to a Shader Resource View. This buffer is available through a template in HLSL and we can read the values by using the Load() method:

```
Buffer<float3> myBuffer;
…
Float3 weight = myBuffer.Load(x);
```

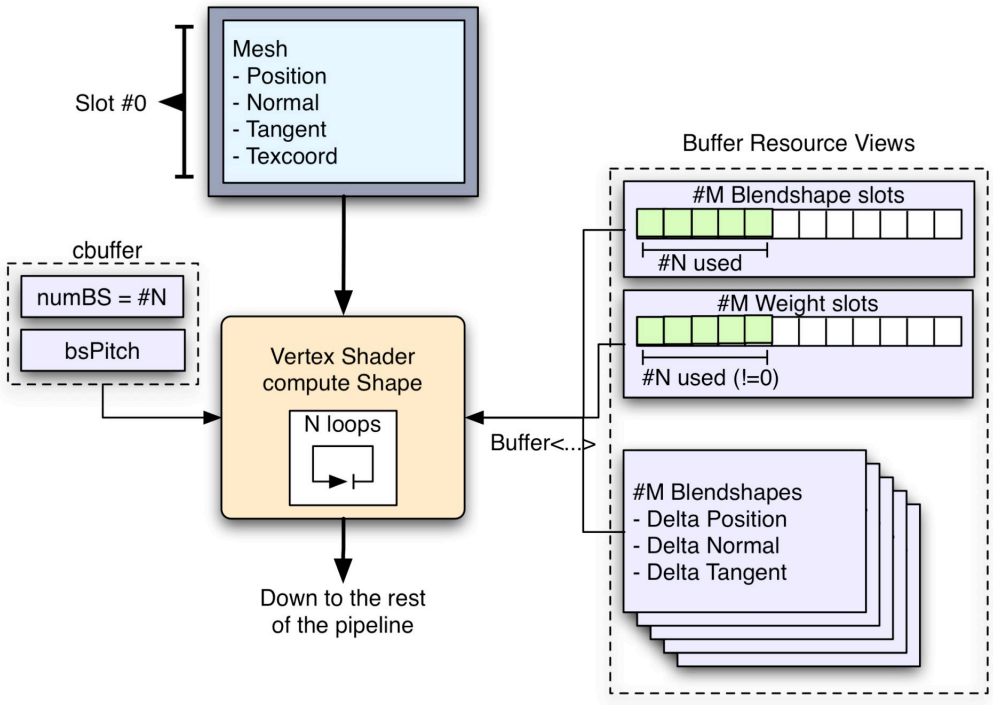
On the C++ side, we create a buffer of data and associate a Shader Resource View with it. Then we can assign it to the HLSL variable as follows :

```
ID3D10EffectShaderResourceVariable *v;
…
v->SetResource(myRV);
```

This method is using these buffers to store all the blend shapes in one single big buffer. Therefore it is possible to access them if we know their index number and the pitch value (the size of these blend shapes).

It is important to exclude from the computation all the blend shapes that aren't used: the ones that have a weight set to 0. We use two other buffers in which we dynamically update the blend shapes' indices and weights. A variable (*numBS*) is then updated with the total number of consistent blend shapes.

The vertex shader simply loops over these two arrays of indices and weights; retrieves the corresponding vertex attributes in the blend shape pointed out by the index; and finally adds these contributions to the final vertex.



There is however a limitation in DX10 to be aware of, when we use buffers to read vertex data:

- It is impossible to use a structure of data (for vertex attributes) in the Buffer<> template. Only basic types like float, float3 etc can be used.
- When we create a Shader Resource View, we face the same problem: only the types from the `DXGI_FORMAT` enum are available. There is no way to specify an input layout in a resource view.

This issue is not a problem at all in our case since our blend shapes are made of three float3 attributes (Position, Normal, Tangent). So we can simply declare a buffer of float3 and step into it three by three. However there is a problem if you want to read a set of vertex attributes made of different widths, say, float2 for texture coordinates, float4 for color, etc.

The easiest workaround is to pad the shorter data. For example a float2 texture coordinate will have to be float4 in memory and the shader will use only the first two components. A more complicated workaround would be to read a set of float4 values and to reconstruct the vertex attributes ourselves in the shader.

Running the Sample

The sample shows how to combine 54 facial expressions on the face of the Dawn character. The demo starts with the animation turned on. This animation is simply reading a set of curves to animate a few weights.

We arbitrarily grouped these expressions two by two in 26 groups: you can play with two sliders to vary the 54 expression intensities. Note that if the animation is on, you may not be able to use the sliders that are being animated. However, check out the ones you can change (try Group 6 for example) at the same time the animation is running: this is a good example on how you can combine complex expressions together.

There is a combo box to allow you to switch between the two modes.

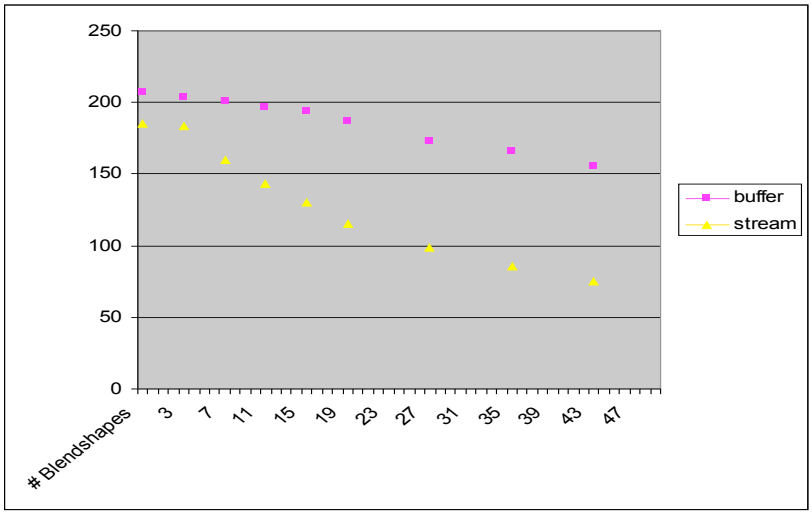
Finally, some optional sliders are also available to change the skin lighting of Dawn.

Performance

Performance varies depending on how many expressions we use.

The first technique using Stream Out buffer is slower than the second using buffer templates and Shader Resource Views. The main reason is because the former is streaming-out the data as many times as needed: the more expressions we have, the more we will loop through the stream buffer, back on the CPU. In the second implementation, everything is performed inside the vertex shader and no intermediate data needs to be streamed out from the GPU pipeline.

The gap in performance between the methods increases as we add more blend shapes. With 6 blend shapes the second method is already 1.34X faster than the first one. When we reach 50 blend shapes used at the same time, the second method is 2.4X faster.



References

- Animation in the "Dawn" Demo. Curtis Beeson. GPU Gems 2004.

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