SparseMorphTargets Sample

Collapse All

This sample implements mesh animation using morph targets.

Path

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

Morph Targets

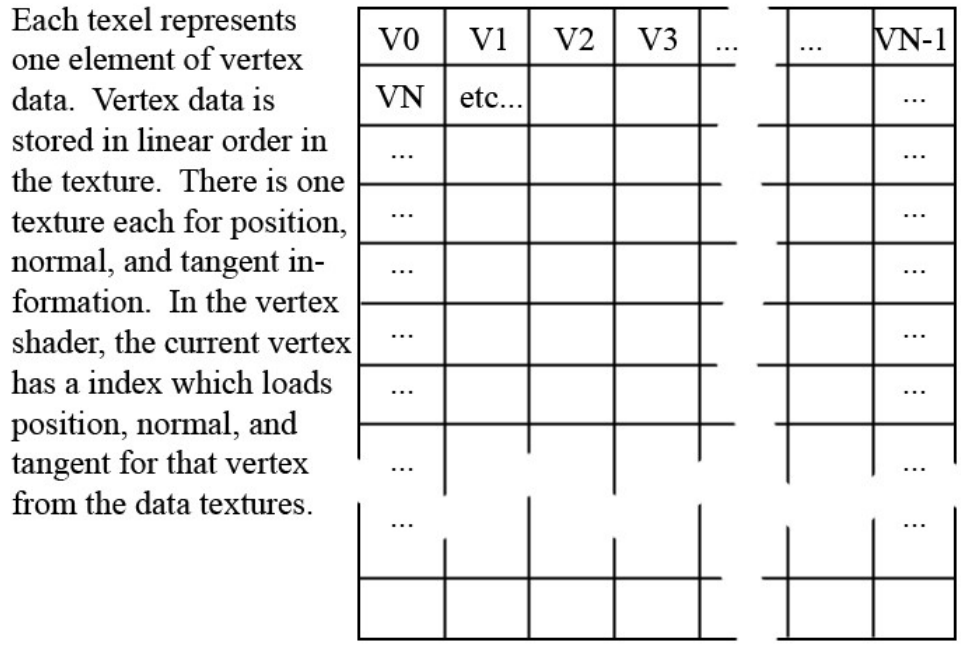
The SparseMorphTargets sample demonstrates facial animation by combining multiple morph targets on top of a base mesh to create different facial expressions.

Morph targets are different poses of the same mesh. Each of the minor variations below is a deformation of the base mesh located in the upper left. By combining different poses together at different intensities, the sample can effectively create many different facial poses.

How the Sample Works

Preprocessing

In a preprocessing step, the vertex data for the base pose is stored into three 2D textures. These textures store position, normal, and tangent information respectively.



This base texture is stored into the .mt file along with index buffer data and vertex buffer data that contains only texture coordinates. Then, for each additional morph target, the mesh is converted to three 2D textures. The base mesh textures are subtracted from the three morph target textures. The smallest subrect that can fully contain all of the differences between the two textures is then stored into the .mt file. By only storing the smallest texture subrect that can hold all of the differences between the two meshes, the sample cuts down on the storage requirements for the morph targets.

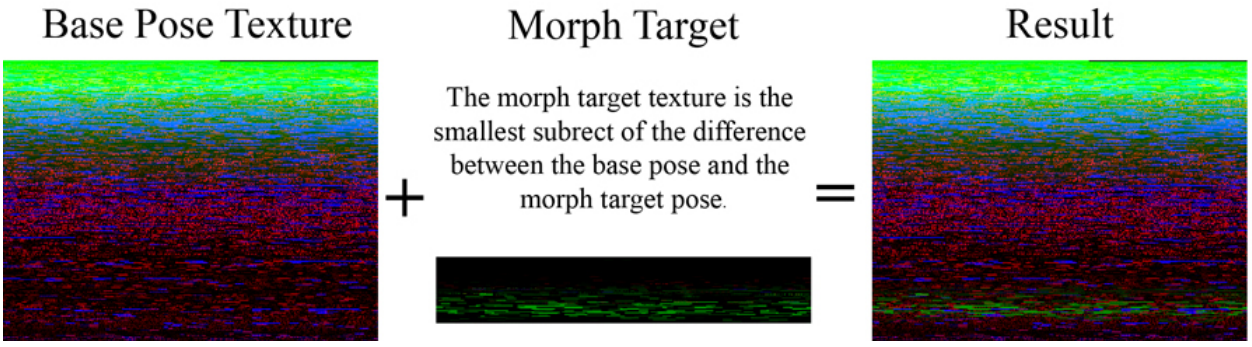
Applying Morph Targets at Runtime

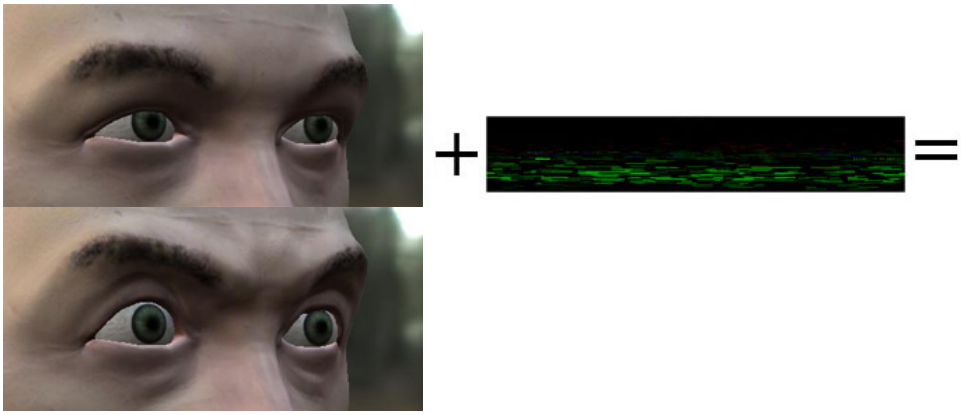
Morph targets are handled in the MorphTarget.cpp and MorphTarget.h files. These contain classes that help with the application of the morph targets. To apply the different morph targets to the mesh, the sample sets up a texture2Darray render target with three array indices. The first holds position data, while the second and third hold normal and tangent information respectively.

The first part of the process involves filling the render targets with the position, normal, and tangent information from the base pose. Then for any morph targets that need to be added, the following occurs:

- The alpha blending mode is set to additive blending
- The viewport is set to the exact size of the pos, norm, and tangent render targets
- A quad covering only the pixels in the render target that will change for the given morph target is drawn
- This quad is drawn with the morph target position, normal, and tangent texture bound as a textured2darray
- A special Geometry Shader replicates the quad three times and sends one to each render target
- The Pixel Shader sets the output alpha to the blend amount. The blend amount is the amount of influence this morph target should have in the final image.

Effectively, the above uses the alpha blending hardware of the GPU to add up a series of sparse morph targets to a base mesh. The position, normal, and tangent render targets now contain the vertex data of the final deformed mesh. These are bound as inputs to the technique that renders the final morphed mesh onscreen.





Rendering

The input vertex stream contains a vertex reference that tells the VSRefScenemain shader which texel in the position, normal, and tangent maps contains our vertex data. In the shader, the uiVertexRef is converted to a 2D texture coordinate. The position, normal, and tangent data is then loaded from textures and transformed as they would be as if they had been passed in via the vertex stream.

```
//find out which texel holds our data
uint iYCoord = input.uiVertexRef / g_DataTexSize;
//workaround for modulus
uint iXCoord = input.uiVertexRef - (input.uiVertexRef/g_DataTexSize)*g_DataTexSize;
float4 dataTexcoord = float4( iXCoord, iYCoord, 0, 0 );
dataTexcoord += float4(0.5,0.5,0,0);
dataTexcoord.x /= (float)g_DataTexSize;
dataTexcoord.y /= (float)g_DataTexSize;

...

//find our position, normal, and tangent
float3 pos = tex2Darraylod( g_samPointClamp, g_txVertData, dataTexcoord ).xyz;
dataTexcoord.z = 1.0f;
float3 norm = tex2Darraylod( g_samPointClamp, g_txVertData, dataTexcoord ).xyz;
dataTexcoord.z = 2.0f;
float3 tangent = tex2Darraylod( g_samPointClamp, g_txVertData, dataTexcoord ).xyz;

//output our final positions in clipspace
output.pos = mul( float4( pos, 1 ), g_mWorldViewProj );
```

Adding a Simple Wrinkle Model

To add a bit of realism the sample uses a simple wrinkle model to modulate the influence of the normal map in the final rendered image. When rendering the final morphed image, the Geometry Shader calculates the difference between the triangle areas of the base mesh and the triangle areas of the current morphed mesh. These differences are used to determine whether the triangle grew or shrank during the deformation. If it shrank, the influence of the normal map increases. If it grew, the influence of the normal map decreases.

Adding Illumination Using LDPRT

Local Deformable Precomputed Radiance Transfer is used to light the head. The PRT simulation is done on the base mesh and the 4th order LDPRT coefficients are saves to a .ldprt file. The sample loads this file into memory and merges it with the vertex buffer at load time. The bulk of the LDPRT lighting calculations are handled by GetLDPRTColor. The current implementation only uses 4th order coefficients. However, uncommenting the last half of GetLDPRTColor, re-running the PRT simulation to generate 6th order coefficients, and changing dwOrder to 6 in UpdateLightingEnvironment may give a better lighting approximation for less convex meshes.

For a more detailed explanation of Local Deformable Precomputed Radiance Transfer, refer to the Direct3D 9 LocalDeformablePRT sample or the PRTDemo sample.