MultiAnimation Sample

Collapse All

This sample demonstrates mesh animation with multiple animation sets using high-level shader language (HLSL) skinning and the D3DX animation controller. The animation controller blends animation sets together to ensure a smooth transition when moving from one animation to another.

Path

Source	*SDK root*\Samples\C++\Direct3D\MultiAnimation
Executable	*SDK root*\Samples\C++\Direct3D\Bin*x86 or x64*\MultiAnimation.exe

Overview

This sample shows how an application can render 3D animation utilizing D3DX animation support. D3DX has APIs that handle the loading of the mesh to be animated, as well as the blending of multiple animations. The animation controller supports animation tracks for this purpose, and allows transitioning from one animation to another smoothly. The sample is divided into two parts: the animation class library and the application.

The animation class library is a general-purpose library that sits between the application and D3DX. It encapsulates the details of loading the mesh from a .x file and manipulating its frame hierarchy to prepare it for rendering, as well as rendering the animated mesh using a vertex shader and a matrix palette for indexed skinning. It is designed to be reusable and customizable.

The application portion contains code specific only to this sample. It uses the animation class library to create instances of the Tiny character, which are animated according to the action they perform. Each instance can be controlled by either the user or the sample. This portion also handles collision detection among instances of Tiny, out of bound detection, and behavior management for instances that are controlled by the sample.

The Application

The application portion of this sample consists of two parts. The first part is the CTiny class. It handles the behavior of the animated mesh in the sample loaded from the tiny_4anim.x file. The second part contains CMyD3DApplication and all of the code that is normally generated by the DirectX application wizard.

In this sample, Tiny can be controlled by either the user or the application (default). When Tiny is controlled by the application, the following happens: First, a random location on the floor is chosen and a move speed (run or walk) is determined. Next, Tiny is turned at the current position toward the new destination and is moved in that direction. After Tiny arrives at the new destination, there is a brief period of idle time before another location is selected where to move Tiny, and the whole process repeats. There can be more than one instance of Tiny. Collision detection is done for the instances, so they can block each other's movement.

The CTiny class has a CAnimInstance member, a class defined by the animation library. CTiny uses this member and its animation controller to perform the animation tasks necessary. In this sample, Tiny has three sets of animation: Loiter, Walk, and Run. The animation controller of Tiny supports two animation tracks. Most of the time, one of these two tracks is active with an animation set. When Tiny's action calls for a change of animation (for instance, going from loitering to walking, or stopping from running), enable the second track with the new animation set to play and set a transition period to complete the transition from the first track to the second. Then, as time is advanced, the animation controller will generate the correct frame matrices that reflect the transition smoothly by interpolating between the two tracks. After the transition period passes, the first track is disabled, and the second track plays the new animation in which Tiny is seen.

CTiny also takes advantage of the callback system to play the footstep sound at appropriate instances in the animation. At initialization time, CTiny sets up a structure, CallbackDataTiny, containing the data to pass to the callback handler. The structure has three members: m_dwFoot indicates which foot is triggering the sound, m_pvCameraPos points to the camera position when the callback is made, and m_pvTinyPos points

to the world space coordinates of Tiny. This data is used to determine which sound to play (left or right), and how loud it should be. The sample defines a class called CBHandlerTiny, derived from **ID3DXAnimationCallbackHandler**, that contains the callback handler function to be called by the animation controller. The callback handler function, HandleCallback, retrieves the data passed as a CallbackDataTiny structure then plays a DirectSound buffer with the appropriate volume based on the values in the CallbackDataTiny structure.

In FrameMove, the sample iterates through the array of CTiny instances and calls Animate on each one of them. This updates the behaviors of all instances, changing them if needed.

The rendering code of the application is relatively simple. It first sets up the view and projection matrices based on the camera position and orientation, then it renders the mesh object representing the floor. Next, it goes through the array of Tiny instances and calls AdvanceTime and Draw on each one of them. AdvanceTime makes the animation controller update the frame hierarchy's matrices, then Draw renders the instance using the updated matrices. Finally, the code renders the informational text as necessary.

The Animation Class Library

The library consists of the following structures and classes:

- CMultiAnim
- CAnimInstance
- CMultiAnimAllocateHierarchy
- MultiAnimFrame Structure
- MultiAnimMC Structure

CMultiAnim

This class is the heart of the library. Its function is to encapsulate the mesh hierarchy loaded from a .x file. It can also create instances of the animating mesh, of the type CAnimInstance, that share the mesh hierarchy from the .x file. These instances are associated with the CMultiAnim object that creates them.

In its initialization method, CMultiAnim::Setup, CMultiAnim determines the maximum size of the matrix palette based on the version of the vertex shader present. It then loads the mesh hierarchy from the given .x file and creates an effect object from the given .fx file that contains the vertex shader with which the mesh is rendered. The frames of the mesh hierarchy and the animation controller are created in this process.

The frames consist of a tree structure representing the hierarchy of the bones, as well as the mesh objects. This structure is shared by all associated instances.

The animation controller is associated with the mesh hierarchy frames. The animation controller that CMultiAnim holds is not used for animation. Instead, when a new animation instance is created, the animation controller is cloned and the new animation controller is owned by the new instance. The application uses an instance's own animation controller to control its animation. Because each instance has its own copy of animation controller, it can animate independent of all other instances.

When the **ID3DXAnimationController::AdvanceTime** method of an animation controller in an animation instance is called, the animation controller will update the transformation matrix for each frame in the mesh hierarchy. The frames are then used in the rendering of the instance.

CAnimInstance

This class represents an animation entity, or an animation instance. Each animation instance has its own animation controller, which allows the instances to be animated independent of each other. Each instance is also associated with a CMultiAnim object, which holds the mesh hierarchy that is shared by all associated instances. The following table lists the methods that are the most interesting for animation control and which are used extensively by the application when animating and rendering.

Method Name	Description
GetAnimController	Returns the animation controller for this instance, of type **ID3DXAnimationController**. With the animation controller, the application can set the animation it needs to play.
SetWorldTransform	Sets the top level world transformation matrix for this instance. When the animation time advances, each frame in the mesh hierarchy is recursively transformed by this

	world transformation to bring it to the correct world space coordinate and orientation.
AdvanceTime	Advances the local animation time for this instance with an optional callback handler. The time advancement causes the animation controller to update the bone positions for this instance. The animation controller will also invoke the callback handler if the keyed event is reached.
ResetTime	Resets the local time for this instance.
Draw	Renders this instance with the HLSL vertex shader. This is normally called after calling AdvanceTime so that the frames are set up properly for rendering.

When an animation instance is rendered, it uses the effect object of its associated CMultiAnim to set the rendering parameters, including the vertex shader. The vertex shader function is responsible for skinning the mesh, transforming the position to screen space, and computing the color. The skinning portion is done by the function VS_Skin. This function is located in the vertex shader file Skin.vsh. The file contains an array of float4x3 named amPalette, which represents the matrix palette for skinning meshes, and the VS_Skin skinning function. VS_Skin is designed to be invoked by another vertex shader function, so an application can call it in its own vertex shader function to handle the skinning process. VS_Skin takes the object space position and normal, up to three blending weights (the last blending weight is computed by subtracting the sum of all other weights from 1), and up to four indices for the matrix palette. The function then transforms the position and normal up to four times with the corresponding matrices in the matrix palette and blending weights, and adds the result together to form the world space position and normal.

CMultiAnimAllocateHierarchy

This class inherits from **ID3DXAllocateHierarchy**. Its purpose is to handle the allocation and deallocation of resources during the loading and releasing of a mesh hierarchy. In this sample, CMultiAnimAllocateHierarchy initializes all members of new frames and mesh containers in CreateFrame and CreateMeshContainer called during the mesh hierarchy creation. In DestroyFrame and DestroyMeshContainer, it frees up all resources allocated in CreateFrame and CreateMeshContainer, respectively.

CreateMeshContainer does a little more work than simply allocating and copying. It also has to do the following:

- Create all of the textures used by the mesh.
- Initialize the mesh container's bone offset array, given by pSkinInfo.
- Obtain the palette size that the mesh has to work with, and call ConvertToIndexedBlendedMesh to create a working mesh that is compatible with the palette size.

MultiAnimFrame Structure

This structure is derived from **D3DXFRAME**. It represents a single frame, or bone, in a mesh hierarchy. A frame may contain siblings or children. The entire mesh hierarchy is structured similar to a tree. The application can add members to a structure derived from **D3DXFRAME** as necessary. In this sample, no additional members are needed.

MultiAnimMC Structure

This structure is derived from **D3DXMESHCONTAINER**. It contains a mesh object attached to the mesh hierarchy along with relevant data for the mesh. A mesh hierarchy can contain any number of mesh containers. In this sample, additional members are defined as listed in the following table.

Type	Name	Description
LPDIRECT3DTEXTURE9 *	m_apTextures	Array of Direct3D texture objects used by the mesh.
LPD3DXMESH	m_pWorkingMesh	A copy of the mesh compatible with the matrix palette size available.
D3DXMATRIX*	m_amxBoneOffsets	Array of matrices representing the bone offsets. This information is obtained from the pSkinInfo member, copied here for convenience.

D3DXMATRIX**	m_apmxBonePointers	Array of pointers to matrix. They point to the TransformationMatrix of various frames of this mesh hierarchy. This array provides an easy mapping from bone index to bone matrix.
DWORD	m_dwNumPaletteEntries	Size of the matrix palette that this mesh uses when rendering. The size cannot exceed the maximum palette size allowed by our vertex shader due to limited registers, and it also cannot be larger than the number of bones in this mesh.
DWORD	m_dwMaxNumFaceInfls	Maximum number of bones that may affect a single face. This value is obtained from **ID3DXSkinInfo::ConvertToIndexedBlendedMesh**. The vertex shader needs this value when rendering in order to know when to compute the last weight.
DWORD	m_dwNumAttrGroups	Number of attribute groups in working mesh. An attribute group is a subset of the mesh that can be drawn in a single draw call. If the working palette size is not large enough to render the mesh in its entirety, it needs to be broken down into separate attribute groups. This is done by **ID3DXSkinInfo::ConvertToIndexedBlendedMesh**.
LPD3DXBUFFER	m_pBufBoneCombos	Contains the bone combination table, in the form of an array of **D3DXBONECOMBINATION** structures. There is one **D3DXBONECOMBINATION** for each attribute group. It identifies the subset of the mesh (vertices, faces, and bones) that can be drawn in a single draw call.

User's Guide

The following general controls are defined in the sample.

Key	Action
Q	Move camera down.
E	Move camera up.
W	Move camera forward.
A	Move camera left.
S	Move camera backward.
D	Move camera right.
N	Next view
P	Previous view
R	Reset camera
F2	Bring up Direct3D Settings menu.
Alt+Enter	Toggle full screen.
Esc	Quit

The following controls are valid when you are viewing a specific mesh instance.

Key	Action

C	Take control
W	Move forward
A,D	Turn
W+Shift	Hold for run mode

Pitfalls and Alternatives

Some of what this sample does can be done differently, with different trade-offs.

Because all instances of Tiny share the same frame hierarchy matrices, the sample must advance time on a specific instance and render it before moving on to another instance. Generally, this does not present a problem. However, if an application wishes to update the matrices for all of the instances first then render them, it can make new animation controllers point to a different set of matrices when cloning. This way, each animation controller has its own set of matrices to work with, and overwriting will not occur, at the expense of more memory usage.

It is also worth noting that the matrix palette in the skinning vertex shader is an array of matrices, and naturally takes up a significant number of constant registers. An application may experience the problem of insufficient constant registers for other use. The application can work with a smaller-sized matrix palette by setting the MATRIX_PALETTE_SIZE_DEFAULT shader #define to a smaller value when creating the effect object in CMultiAnim. The drawback of this approach is that the mesh may contain more subsets that need to be drawn separately.

Note that this sample specifically demonstrates rendering animated meshes. It does not handle static meshes, although the code can be extended to do that.

Another task that the sample only does partially is the complete preservation of animation controllers' states when the Direct3D device is released and re-created. This generally happens when the application switches from HAL to REF device (see **Device Types (Direct3D 9)**), or vice versa, or from one Direct3D device to the other on a dual-monitor system. In MultiAnimation, when the Direct3D device is released, all animation controllers are also released. The animation controllers are then re-created when the sample initializes the new Direct3D device. This presents a problem, because animation states stored in animation controllers are lost. The sample loses the knowledge of what animation each track is playing, what tracks are enabled, and the speed and weight of each track before the old device is released. To remedy this issue, an application should, before releasing Direct3D objects in its cleanup function (DeleteDeviceObjects for MultiAnimation), retrieve the animation controllers' states and save them in a buffer. Then, after it re-creates the animation controllers that it needs (in InitDeviceObjects for MultiAnimation), it should restore the animation controllers' states from the buffer. For each animation controller, the following states should be saved.

- Current time of the animation controller, obtained with **ID3DXAnimationController::GetTime**.
- Name of the animation set that each track is playing. Obtain this information by calling **ID3DXAnimationController::GetTrackAnimationSet**, then **ID3DXAnimationSet::GetName**.
- Track description of all tracks. Use **ID3DXAnimationController::GetTrackDesc** to obtain this description.
- Current events for all tracks, if there are events running. This can be retrieved with **ID3DXAnimationController::GetCurrentTrackEvent** and **ID3DXAnimationController::GetEventDesc**.
- All keyed track events. To receive every event keyed, first call **ID3DXAnimationController::GetUpcomingTrackEvent** with hEvent set to NULL. This returns an event handle to the keyed event to happen next. Then, pass the handle to **ID3DXAnimationController::GetEventDesc** to obtain the description for this event. After that, take the handle received from the previous **ID3DXAnimationController::GetUpcomingTrackEvent** call and call **ID3DXAnimationController::GetUpcomingTrackEvent** again with it. This gives you the handle to the second keyed event. Repeat this process to receive all keyed events.
- Current priority blend for the animation controller, by calling **ID3DXAnimationController::GetPriorityBlend**.
- Current priority blend event, if one is running, by calling **ID3DXAnimationController::GetCurrentPriorityBlend** and **ID3DXAnimationController::GetEventDesc**.
- All keyed priority blend events. This is done by calling

ID3DXAnimationController::GetUpcomingPriorityBlend and **ID3DXAnimationController::GetEventDesc**. The way to handle all events is similar to handling keyed events in the tracks.

For reasons of simplicity, MultiAnimation does not preserve all of its animation controllers' states, but rather it saves only the animation set that its current track is playing. Consequently, if the Direct3D device object is released and re-created during an animation transition, the animation will not appear the same after the transition as before; it will appear as if the transition has been completed. In addition, all keyed events will not be restored after re-initialization.