Workbook Ten:
Animation

Lab Project 10.1: Adding Animation to CActor

In Lab Project 9.1 we implemented a CActor class that encapsulated the loading and rendering of hierarchical multi-mesh representations. In this lab project we are going to continue to evolve CActor by adding animation support for its internal frame hierarchy. As each actor represents a single hierarchy loaded from an X file, the actor will become the perfect base for modeling game characters as well as other animated entities. Each character is contained in its own frame hierarchy inside the actor and will have its own animation controller to play back its animations via the CActor interface. We must also remember that a single CActor object could also be used to animate an entire scene if all the meshes and animations in that scene were loaded from a single hierarchical X file.

In the Chapter 10 textbook, we learned that D3DX exposes an easy-to-use comprehensive animation system. We will now take the information we learned in that lesson and wrap it up inside of our CActor class. Lab Project 10.1 will use the new extended CActor object to load an animated scene from an X file and play it back in real-time. Figure 10.1 shows the code from lab project 10.1 in action.

Figure 10.1

The X file we will load contains a small futuristic scene with many moving meshes, including a space ship that takes off from a launch pad and flies off into the distance. There are also many other moving meshes such as revolving radar dishes and hanger doors that open and close. In this first simple example, a single X file will be loaded by our actor. The file will contain the entire scene and all animation data in a single animation set. Therefore, a single actor will contain all objects in the scene that you see in Figure 10.1.

Because a single animation set will contain the animations for all objects in our scene, playing back this animation could not be easier. The CActor::LoadActorFromX function will call the D3DXLoadMeshHierarchyFromX function to load the X file. This function will automatically create the animation controller and its animation set before returning it to CActor for storage. (The animation controller will be a new member variable that we will add to CActor.) The controller will then be used by the CActor member functions to play the animated scene in response to application requests.

Because the D3DXLoadMeshHierarchyFromX function will (by default) automatically assign the last animation set defined in the X file to the first track on the animation mixer, when the D3DXLoadMeshHierarchyFromX function returns the newly created controller back to the actor, we have absolutely no work to do. The only animation set defined in the X file in this example will have been assigned to the first mixer track on the controller and is ready to be played immediately. All our CActor has do at this point, is call the controller's AdvanceTime method with every requested update to generate the new frame matrices for the hierarchy. For this reason, Lab Project 10.1 is an excellent initial example of using the animation system. It clearly demonstrates the ease with which cut scenes or scripted sequences can be played out within a game engine. Notice that as the animation sequence plays out, the player still has freedom of movement within the scene. This allows the player to feel part of a dynamic scene instead of simply watching a pre-rendered sequence.

Although Lab Project 10.1 will only use a very small subset of the D3DX animation system's overall features, we will still implement all the animation functionality exposed by D3DX in CActor in this project. This will prevent us from having to continually revisit CActor later in the course when we wish to use animation features that have not previously been coded. This will mean of course that we will have to add many animation methods to our CActor interface so that an application can access all of the features offered by D3DX. For example, our CActor must provide the ability for an application to assign animation sets to tracks on the mixer, configure the properties of those mixer tracks, and allow the application to reset the global timer of the animation controller. The CActor interface must also expose methods that allow the application to register events with the event sequencer on the global timeline as well as expose the ability for scheduling of callback keys for a given animation set. While this sounds complicated, most functions will be simple wrappers around their ID3DXAnimationController counterparts. Once we have this task out of the way, we will have a CActor class that is reusable in all future applications that we develop. This will be a powerful tool indeed and will allow us to add animated objects to our scenes with ease in the future. Furthermore, we will need this animation system in place in the following chapter when we move on to the subject of character skinning and skeletal animation (something else that will be managed by our CActor class).

Expanding the Limits of the Controller

Another feature we would like our actor to expose to the application is the ability to easily upgrade the maximum limits of the controller. In the textbook we learned that the maximum limits of the animation controller must be specified at controller creation time. This is certainly a restraint we could put up with if we were always manually building our animation controllers, but when using the D3DXLoadMeshHierarchyFromX function, the controller is created for us and we are afforded no control over the maximum values. For example, the controller created by

D3DXLoadMeshHierarchyFromX will always have two mixer tracks. This may not be enough to satisfy the blending desires of our applications. Additionally, the maximum number of animation sets that can be simultaneously registered with the controller will always be equal to the number of animation sets defined in the X file. Once again, this is often acceptable, but sometimes our application may wish to programmatically build additional animation sets once the X file has been loaded, and then register these new sets with the controller. The controller returned from D3DXLoadMeshHierarchyFromX would have no room for these animation sets to be registered, so we will have to account for this limitation.

We will address this problem by exposing a method called CActor::SetActorLimits. With this function, we can specify the exact number of animations, animation sets, and mixer tracks that we would like our actor's animation controller to support. This function is very flexible because you can call it before or after the X file has been loaded. If you call it before you have call CActor::LoadActorFromX, then the maximum limits you pass in will be stored in the actor's member variables. When the call to CActor::LoadActorFromX is finally made and the animation controller is created, the function will automatically clone the D3DX-generated animation controller into one which matches the desired maximum limits. When CActor::LoadActorFromX returns control back to the application, the actor will contain its frame hierarchy with all animation data intact and an animation controller with the desired maximum limits. We can also call the CActor::SetActorLimits method even after the X file has been loaded and it will still work fine. If the X file is already loaded, the SetActorLimits method will simply clone the current animation controller being used into one with the desired maximums specified by the application.

> **Note:** When we clone the animation controller, we can specify different creation parameters from those used in the original (e.g., a higher number of mixer tracks). All the animation data stored in the original controller will be copied into the cloned controller, ensuring data integrity. Our CActor::LoadActorFromX and CActor::SetActorLimits methods use cloning to allow the application to override the default limits of the animation controller created by the D3DX loading function. This also allows our application to expand the limits of the actor's controller at any time, even when the application is running.

The following code shows how an application might use the CActor class to load an animated X file and then set the limits of the controller:

```
// Create Actor
CActor *pActor=new CActor;

// Register scene call back function to manage mesh resources
pActor->RegisterCallback( CActor::CALLBACK_ATTRIBUTEID, CollectAttributeID,this );

//Load an X file into the actor. All meshes in the managed resource pool
pActor->LoadActorFromX( "MyXFile.x", D3DXMESH_MANAGED, m_pD3DDevice );

// Set the controller so that it can handle AT LEAST 15 animation sets and 10
// blending tracks and capable of registering 150 sequencing events
pActor->SetActorLimits( 15 , 10 , 0 , 150 , 0 );
```

As the previous code demonstrates, changing the limits of the actor, even after its controller has been created, is perfectly safe. We will look at this function in detail later. For now, just know that if we pass in zero for any of the parameters, that property will be inherited from the original controller during the cloning process. For example, the third parameter to this function is where we can specify the number of

animation outputs we would like the actor's controller to handle. This will have already been set in the original controller (by D3DX) such that it can at least animate all the animated frames in the hierarchy. This is usually exactly what we want it to be. Unless we are adding additional frames to the hierarchy programmatically, we will normally not want to alter this value. Therefore, by passing in zero for this property, the SetActorLimits method knows that the new controller it is about to clone should have its maximum animation outputs value remain the same as the controller from which it is about to be cloned. This is useful because the application will not be forced to query the number of animation outputs currently being managed by the actor's controller, simply to pass that same value back into the SetActorLimits method unchanged.

The final parameter to this function is where we can specify the maximum number of callback keys that can be registered with any animation set. We will discuss why this is needed in the following section. It is also worth noting that the application can use the SetActorLimits method whenever it likes. For instance, in the above example we set the maximum number of mixer tracks to 10. However, later in the application we might decide to expand that to 15. We could do so simply by calling the function again with 15 instead of 10 as the second parameter. We could pass zero in for all the other parameters so that the newly cloned controller will inherit all the other properties from the original controller.

The Actor Exposes a One-Way Resize

For reasons of safety and ease of use, we made the design decision that the SetActorLimits method should only ever clone the controller if any of the specified maximum limits exceed those of its current controller. The function will never clone the controller if we specify maximum extents which are less than the current controller's maximums. There are a number of reasons we decided to take this approach.

First, we do not wish to burden the application with having to write code to query the current maximum extents of the controller. It is much more convenient if we can just specify the properties we need (such as the number of mixer tracks) without having to be aware of all the other properties that were setup automatically (e.g., the number of animation outputs). Second, we could accidentally break the controller's ability to animate the data loaded from the X file if we downsized its maximum extents such that it could no longer properly use the information stored in the original X file. For example, if the original controller created by D3DX needed to animate 50 frames in the hierarchy, it would have been created with the ability to manage 50 animation outputs. If we then called the SetActorLimits method and incorrectly set this parameter to some value less than 50, information would be lost in the clone operation and some or all of the frames in the hierarchy that should be animated would no longer be. This would essentially break the entire animated representation created by the artist. In order to avoid this, the application would first need to query the current number of animation outputs being used by the controller and would need to pass no less than this value into the SetActorLimits method. With our 'expand-only' philosophy, this entire burden is removed. We do not even have to know about the properties of the mixer that we have no interest in. We can simply specify that we require no less than a certain value for a given extent.

In the expand-only system, the SetActorLimits method will never clone an animation controller that is incapable of animating the data in the original controller, regardless of the values passed in by the application. This allows the application to call this method with any value without breaking the actor's ability to animate the original data loaded from the file. While this does mean that there may be times when the maximum limits of the controller may be superfluous to your applications requirements, this is often very rare, since the original animation controller will be created with the bare minimum capabilities needed to animate the original X file data. Therefore, you will usually only want to extend these capabilities and not reduce them. One exception to the rule might be if your application only wishes to use a single mixer track. We know from the accompanying textbook that the animation controller created by D3DXLoadMeshHierarchyFromX will always have at least two mixer tracks. In this case, the animation controller would have a mixer track that you have no intention of using so you could downsize. However, the memory taken up by a single wasted mixer track is certainly acceptable given the ease and safety at which the actor can now be used.

A Flexible Callback Registration Mechanism

One of the more frustrating limitations of using the controller's callback system is that the memory for the callback keys must be allocated at animation set creation time. You will recall from the textbook that when creating an animation set manually (using the D3DXCreateKeyframedAnimationSet function), we must specify how many callback keys we would like to register with the animation set and pass in a pointer to an array of callback keys. These callback keys are then allocated and stored inside the animation set. While this is a limitation we can live with when creating animation sets manually, we know only too well that when using the D3DXLoadMeshHierarchyFromX function, the animation controller and all its animation sets are created for us automatically. As it is very unlikely that you would have callback keys stored in your X file, we can assume that under most circumstances the application will want to register callback keys with the controller *after* the X file has been loaded. At this point however, the animation controller will have been created along with its animation sets (with no callback key data assigned to them). We cannot simply add callback keys to an animation set at this time, because the memory for our callback keys will not have been allocated when the animation set was created by D3DX. D3DX would have had no way of knowing that we intended to register anything at all. Suffice to say, cloning will come to the rescue here as well.

The system we designed to handle the post-load registration of callback data does not just involve data cloning. It also incorporates an application-defined callback function which will be registered and called by the actor to allow the application to fill an actor owned temporary callback key array. This callback function will be called by the actor in the CActor::LoadActorFromX function just after the X file has been loaded. The actor will pass the application defined callback function an array of callback keys which the function can fill with callback data. When the callback function returns, the actor will have an array of callback keys for a given animation set. It will now need to replace the respective animation set that was loaded by D3DX with a clone which has room for the callback data. We will not be cloning the animation controller but will instead manually create copies of the animation sets ourselves. That is to say, for each animation set currently registered with our controller, we will create a new animation set with the correct number of callback keys reserved. When we create these animation set copies, we will pass in the callback data that the application defined callback function stored in our temporary callback

key array. Once we have created copies of all the animation sets with the callback data registered, we will un-register and free the original animation sets and register the new ones (with the callback data) in their place.

As callback keys are defined on a per-animation set basis, the application defined callback function will be called once for each animation set loaded from the X file. This will allow the application to register callback data with any of the loaded animation sets. All this will happen inside the CActor::LoadActorFromX function, abstracting away the application from the entire process. All the application has to do is implement the callback function and register it with the actor prior to calling CActor::LoadActorFromX.

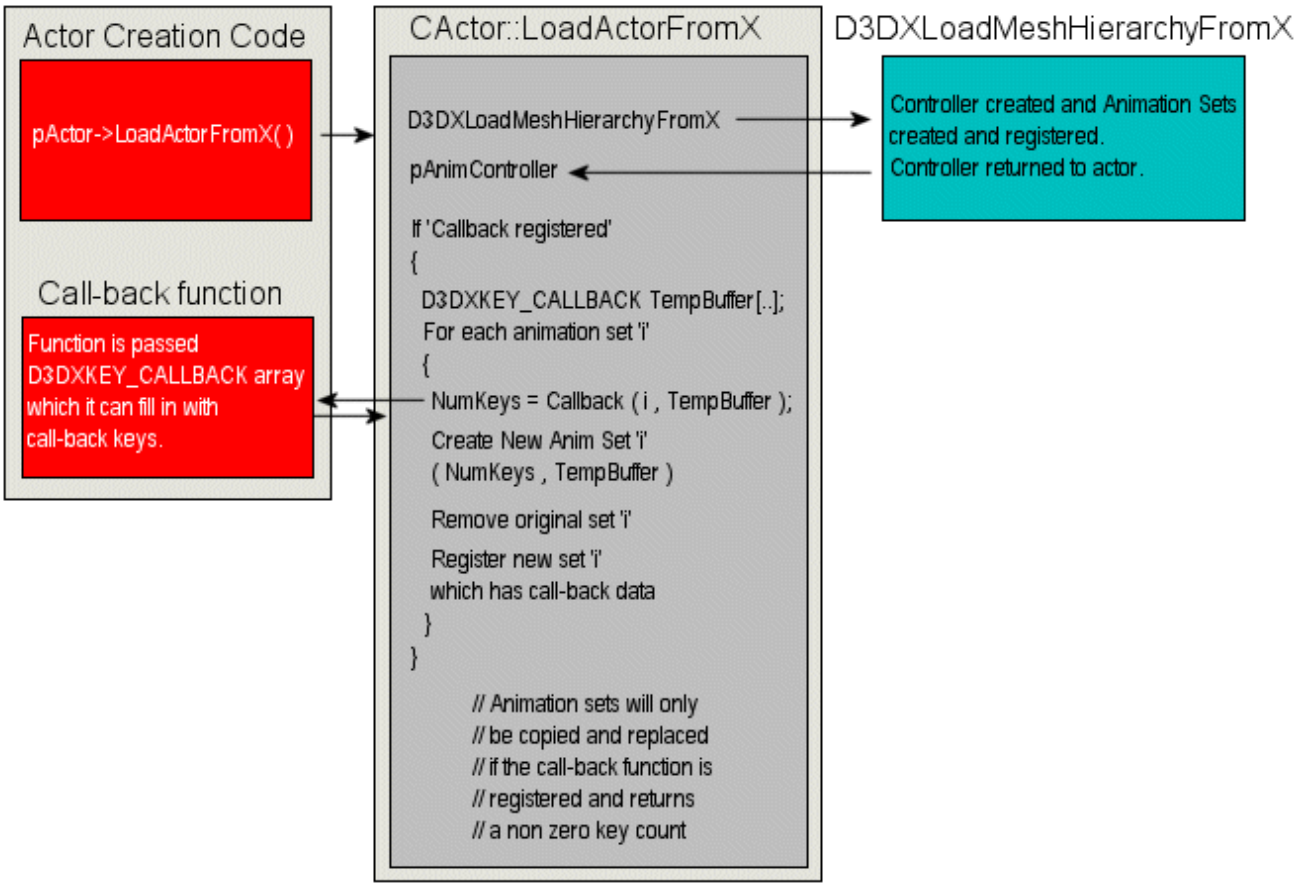

Figure 10.2

Figure 10.2 demonstrates the type of interaction that we will establish between the actor and the application. The application calls CActor::LoadActorFromX, which in turn calls D3DXLoadMeshHierarchyFromX to load the X file. If the X file contains animation data, the actor will be returned a pointer to an animation controller. At this point, if the application has registered a callback key callback function with the actor, it will allocate a temporary array of callback keys. It will then loop through each animation set registered with the controller and call the application defined callback function. This function will be passed the temporary callback key array which it can use to fill with callback keys for that animation set. The function should return an integer describing the number of callback keys it placed in this array. If this is non-zero, the actor will create a new animation set, passing

in this array of callback keys at creation time. The new animation set will not yet be attached to the controller at this point but will contain all the callback keys the application callback function specified for it. The next task will be to copy over all the animation data from the original animation set into the new animation set so that all animation data is maintained. At this point the new animation set will be a perfect clone of the original, with the exception that it now has the callback keys contained within. Finally, we un-register the old animation set from the controller and replace it with the new one. This will be done for each animation set registered with the controller, giving the application the chance to replace any animation sets that need callback key data.

The application defined callback function is also passed a pointer to the original animation set interface. This allows the callback function to check the name of the animation set for which it is being called so that it can determine if it wishes to register any callback keys for this set.

In our implementation, we decided to let the actor allocate the temporary callback key array that is passed into the callback function. One might wonder why we do not just let the callback function return a pointer to an application allocated array. In fact, this would present some problems. First, if the application allocated the array and returned a callback key array pointer to the actor, the programmer might be tempted to generate this array on the stack in the callback function. When the callback function returns the pointer to the actor, the stack information will be destroyed and the actor will have an invalid pointer. Second, even if the callback array is dynamically allocated on the heap in the callback function and returned to the actor, the application will need to implement a cleanup system where this memory is freed when the actor is deleted. This would need to be a little bit like the DestroyMeshContainer callback method that we used during frame hierarchy destruction. We certainly cannot allow the actor to free this application allocated memory in its destructor since it did not allocate this memory itself. It would not know for sure how the memory was allocated and therefore how it should be freed (e.g., we would not want our actor to use ‘delete’ instead of ‘free’ if the application had allocated the key array using malloc).

This is all a bit of a pain for the sake of a simple temporary buffer that is only used to communicate callback keys from the application to the LoadActorFromX function. After all, this temporary buffer is no longer needed once the new animation set has been created because the animation set creation function makes an internal copy of the callback data passed in. Therefore, what we decided to do is simply let the *actor* allocate a temporary buffer which is reused with each call to the callback function. The callback function can simply fill this buffer, which is then automatically accessible to the actor on function return. Furthermore, because the actor allocated this memory, it can release it with confidence after all the callback keys have been registered for each animation set.

Registering a callback key callback function with our actor will be done in exactly the same way we register texture and attribute callback functions. You will recall from the previous lab project that the actor maintains a callback function array (see CActor.h):

```
CALLBACK_FUNC               m_CallBack[CALLBACK_COUNT];
```

Each element of this array defines a single callback function contained in a CALLBACK_FUNC structure (see main.h):

```
typedef struct _CALLBACK_FUNC   // Stores details for a callback
{
    LPVOID  pFunction;          // Function Pointer
    LPVOID  pContext;           // Context to pass to the function

} CALLBACK_FUNC;
```

Each member of this structure contains two void pointers. The first member is where a pointer to the callback function should be stored. The second member is where we can store a pointer to arbitrary data that we would like passed to the callback function. In our projects we always use this to store a pointer to the instance of the CScene class in which the callback function is a static method. This is so the callback function can access non-static members.

You will recall that we store information in this array using the CActor::RegisterCallback function. We used this same function to store texture and attribute callback functions in the actor's callback function array in previous lab projects (see CActor.h):

```
bool RegisterCallback  ( CALLBACK_TYPE Type, LPVOID pFunction, LPVOID pContext );
```

This function (which we covered in a previous lesson) takes the passed function pointer and context pointer and stores them in a CALLBACK_FUNC structure in the actor's callback array. You will hopefully recall that the first parameter of this function should be a member of the CALLBACK_TYPE enumerated type which is defined within the CActor namespace:

```
enum CALLBACK_TYPE
{
    CALLBACK_TEXTURE = 0,
    CALLBACK_EFFECT = 1,
    CALLBACK_ATTRIBUTEID = 2,
    CALLBACK_COUNT = 3
};
```

This enumerated type describes the index in the actor's callback array where a particular callback function should be stored. For example, if we pass CALLBACK_TEXTURE into the CActor::RegisterCallback method, this pointer and context will be stored in the first element of the array. Using this enumerated type, the actor can quickly test the index of its callback array to see if a particular callback function has been registered by the application.

Previously, we only needed callback functions to parse textures and materials during X file loading, but now we need to add a new member to this enumerated type. We must also increase the CALLBACK_COUNT member so that we make room at the end of the actor's callback function array for a new callback function type. This callback function will be called by the actor (if it has been registered) to collect callback keys for animation sets during the loading process.

The updated CALLBACK_TYPE enumeration is now defined in CActor.h as:

```
enum CALLBACK_TYPE
{
 CALLBACK_TEXTURE = 0,
 CALLBACK_EFFECT = 1,
 CALLBACK_ATTRIBUTEID = 2,
 CALLBACK_CALLBACKKEYS = 3,
 CALLBACK_COUNT = 4
};
```

If element [3] in the actor's callback function array is non-NULL, it will contain the pointer to a callback keys callback function. Notice that because we have now set CALLBACK_COUNT to 4, the callback function array will be statically allocated in the actor's constructor to contain four elements instead of three.

So, registering our callback keys callback function will be no different from registering any of the attribute callback functions. In fact, the following example shows how the CScene class might allocate an actor, register an attribute callback and a callback keys callback function, before loading the animated X file into the actor. As you will see, it is simply an additional function call. Remember that the 'this' pointer passed into the CActor::RegisterCallback method (as the context parameter) allows the instance of the scene class access to the non-static member variables within the static function.

```
//Allocate a new actor
CActor * pNewActor = new CActor;

// Register an AttributeID call back ( non-managed mode actor )
pNewActor->RegisterCallback(CActor::CALLBACK_ATTRIBUTEID,CollectAttributeID, this );

// Register callback function to add callback keys to the actor
pNewActor->RegisterCallback(CActor::CALLBACK_CALLBACKKEYS, CollectCallbacks, this );

// Load the X file into the actor
pNewActor->LoadActorFromX( "MyXFile.x", D3DXMESH_MANAGED, m_pD3DDevice );
```

In order for the actor to be able to call the callback function, it must be aware of the function's parameter list and return type. Therefore, when we implement callback functions, we must make sure that we follow the strict rules laid down by the actor. When the CActor calls this new callback function, it will use a pointer of type COLLECTCALLBACKS (defined in CActor.h):

```
typedef ULONG (*COLLECTCALLBACKS ) ( LPVOID pContext,
                                     LPCTSTR strActorFile,
                                     LPD3DXKEYFRAMEDANIMATIONSET pAnimSet,
                                     D3DXKEY_CALLBACK pKeys[]
                                   );
```

This is a pointer to a function that returns an unsigned long (the number of callback keys placed into the passed array by the callback function). The function should also accept four parameters: a void pointer to context data (the instance of the CScene class for example), a string (the name of X file will be passed by the actor here), an ID3DXKeyframedAnimatonSet interface pointer (the animation set currently being processed by the actor which we may wish to register callback keys for), and an empty array of callback keys that the function will place its callback key data into if it wishes to register any for the passed set. We must make sure that our collect keys callback function is defined exactly in this way.

Below, we see how the CollectCallbacks method is defined in CScene.h. This is the callback function which will be registered with the actor for the processing of callback key data. Note how it matches the exact function signature described above. The details of the method itself will be discussed in a moment.

```
static ULONG CollectCallbacks ( LPVOID pContext, LPCTSTR strActorFile,
                                LPD3DXKEYFRAMEDANIMATIONSET pAnimSet,
                                D3DXKEY_CALLBACK pKeys[] );
```

Our callback functions must always be static if they are to be methods of classes. They must exist even if no instance of the class has been allocated. The function implementation would then be started like this:

```
ULONG CScene::CollectCallbacks( LPVOID pContext, LPCTSTR strActorFile,
                                LPD3DXKEYFRAMEDANIMATIONSET pAnimSet,
                                D3DXKEY_CALLBACK pKeys[] )
{
   // fill in array here with key data
}
```

Before we start to discuss how to fill in the callback key array, one thing may be starting to trigger alarm bells. How does our callback function know how big the empty D3DXKEY_CALLBACK array (passed in by the actor) is going to be so that it does not try and store more keys than the array has elements? Furthermore, this temporary array has been allocated by the actor before being passed to us. So how does the actor have any idea how much memory to allocate for this array? It would seem to have no means for knowing how many callback keys we might want to register for a given animation set.
But this is not the case. When we discussed the SetActorLimits method earlier, we deferred discussion of its final parameter, simply saying that this was where we specified the maximum number of callback keys that can be registered with a single animation set. This setting is unlike the other properties that we set via SetActorLimits in that it does not alter any properties of the controller -- it simply informs the actor how big to allocate this temporary callback array that will be passed into the callback function during loading.

For example, if we specify (via SetActorLimits) a value of 20 for the maximum callback key count, then a temporary array of 20 D3DXKEY_CALLBACK structures will be allocated in the LoadActorFromX method. This temporary array will be passed into the callback function each time it is called (for each animation set loaded). It stands to reason that we would never want to try and store more than twenty keys in the passed array in this example for a given animation set. As this same temporary buffer is used for key collection for all animation sets, it also stands to reason that the callback function must return the number of key's it placed into the array. If we decide for example, that we would like to register 10 keys for animation set 1 but only 5 with animation set 2, we would set the actor limits so that it had a maximum of 10 callback keys per animation set. When the actor was loaded, the callback function would be called twice. Both times an array of 10 elements would be passed that can be filled by the callback function for each set being processed. The second time the function was called (for animation set 2) the callback function would only use 5 out of the 10 possible array elements. Therefore, the return value of the function correctly informs the actor how many keys out of the possible 10 (in this example) were actually used for a given animation set. This value is then used to reserve space for callback keys

in the newly cloned animation set and the callback data copied. The temporary buffer is no longer needed and can be freed once all animation sets have been cloned.

Of course, the important point is how we might implement such a callback function. While we know that the D3DXKEY_CALLBACK structure is a simple structure that contains only two members (a timestamp and a void pointer to context data), it is exactly in the setting of this context data that certain problems arise which must be overcome. This leads on to our next section.

Safe Callback Data

One of the problems with registering context data with a D3DXKEY_CALLBACK structure occurs when the animation controller is released. Let us say for example, that we implement a callback key callback function that simply sets one callback key for an animation set called “Tank” inside an X file called “MyXFile.x”. We will assume that we wish to assign an arbitrary structure (which we shall call MyStruct) as the context data to this callback key. The members of this imaginary structure are not important -- it might contain sound effect data or a series of properties that need altering on the mixer track when this event occurs -- for this example. A naïve first approach might be to implement the callback registration function like so:

```
ULONG CScene::CollectCallbacks( LPVOID pContext, LPCTSTR strActorFile,
                                LPD3DXKEYFRAMEDANIMATIONSET pAnimSet,
                                D3DXKEY_CALLBACK pKeys[] )
{

    // Is it the right actor
    if ( _tcsicmp( strActorFile, _T("MyXFile.x") ) != 0 ) return 0;

    // Make sure we only assign it to the correct animation set
    if ( _tcsicmp( pAnimSet->GetName(), _T("Tank") ) != 0 ) return 0;

    // Allocate an arbitrary structure
    MyStruct * pData = new MyStruct;
    MyStruct->A = some data goes here
    MyStruct->B = some data goes here
    MyStruct->C = some data goes here
    MyStruct->D = some data goes here


    // store the data object in the call back structure along with the timestamp
    pKeys[0].Time          = 150.0;
    pKeys[0].pCallbackData = (LPVOID)pData;


    // Notify that we have one key
    return 1;
}
```

While this function makes a glaring error (which we will discuss in moment), it does show how the callback function can be used to add a key to a specific animation set. This function, when called by the actor during the loading process, would be returned a single callback key for the animation set.

This function will be called by each actor for which this function is registered. Therefore the first thing we do is test to see whether this function is being called by the right actor. In this example we are only interested in the actor created from the X file called 'MyXFile.x'. As the function is passed the name of the actor (which is the same as the X file it was created from), we use the _tcsicmp method to compare the name of the actor against the string of the X file we are interested in. If there is no match, then this function has been called by an actor that we are not interested in registering callback data for, so we exit.

Next we test the name of the passed animation set against the string "Tank", as we are only interested in registering a callback key for the Tank animation set. This is why it is very useful that the interface of the animation set for which the function is being called for is also passed in by the actor. It allows us to use the ID3DXKeyFramedAnimationSet interface to query the animation set to determine if it is an animation set we are interested in. Remember, although at this point in the function we know we have the actor we are looking for, this function will be called by that actor for every animation set it contains. As we are only interested in one of these animation sets, we perform the string comparison between the name of the animation set passed in and the name of the animation set we are looking for and return from the function if no match is found.

Finally, we allocate some arbitrary structure on the heap, assign it some arbitrary data and assign its pointer to the first callback key in the callback key array. A pointer to this structure will be passed to the callback handler passed into AdvanceTime, which may contain data such as the name of a sound effect to play. In this example we have set the timestamp of the callback key to 150 ticks. What these ticks mean in seconds is dependant on the ticks per second ratio of the animation set. If you wish to find that out so that you can convert the second value into a tick value, there is no problem there. Because the function is passed the ID3DXKeyframedAnimationSet interface, we can call its GetSourceTicksPerSecond method and use the returned value to map from one to the other.

So, we have a good idea now how our callback key callback function works. Obviously, this function might grow quite large if you wish to assign lots of different callback keys to lots of different animation sets during the loading process. But even in that case, the system is quite simple. So what was wrong with the above example?

The problem is tied into the procedure of destroying the actor and its underlying animation controller. When we destroy the actor, the actor releases its controller. The controller in turn releases each of its animation sets until it has all been unloaded from memory. The problem is that when the animation set is released, its callback key array is destroyed also (just as we would expect). However, in the above example, we allocated a structure on the heap inside the registration function and stored its address in the callback key's context data pointer. If the callback key structure is simply released, the only surviving pointer to that structure will be deleted also. We now have no way of freeing the memory of the context data structure that was allocated in the callback registration function. When the application terminates, we will have memory leaks, especially if lots of callback keys are being used which have had structures allocated for their context data in this way. This is an easy problem to fix, but we wish to use a solution that is elegant.

One approach might be to fetch all callback key structures from each animation set inside the actor's destructor before it releases its animation controller. We know that the ID3DXKeyframedAnimationSet interface exposes a GetCallback method to allow the controller to step through its list of callback keys. We can simply use this same function to get at the context pointer of each callback key in the animation set's callback key array. Before releasing the controller, we would loop through each animation set, and for each set, repeatedly call its GetCallback method until we have fetched all of its callback keys. You will recall from the textbook that each call to this method returns the time of the callback key and its context data pointer. We can use these context pointers to delete the structures that have been allocated and assigned to this pointer in the callback registration function.

But there is a problem with this approach too. When we fetch the callback context pointer (via the GetCallback method) we are returned a void pointer to the context data. We have no idea how the application allocated it or even what the pointer is pointing at. It could be the address of a structure or class allocated on the heap, in which case releasing it is exactly what we want to do. On the other hand, it could be the address of a constant global variable that was statically allocated. We would cause an error if the actor tried to release something like that. The actor simply has no idea what the context pointer of a callback key is pointing to and how the data it points to was allocated. Therefore, there is no way it can safely make any assumptions about how to delete it (or even if it should be deleted).

Now you might be thinking that one way around this would be to avoid forcing the actor to release this memory at all. Instead, when we allocate the structure inside the callback function, we could add a copy of its pointer to some CScene array. Therefore, even when the actor is deleted and all callback key arrays have been deleted, the scene object would still have the array of all the context data pointers which could be deleted in a separate pass when the application is shut down. This is not very nice though since we should not be happy with the application having to implement its own container class simply to handle a process that should be performed by the actor. Furthermore, we may wish to assign many different types of structures (as context data) to different callback keys, and they may not all be able to be derived from the same base class. In short, the application may have to maintain multiple arrays of callback key context data pointers for the various types that may be used. This approach is ungainly and simply will not do.

COM to the Rescue

If we think about the way that COM was designed, one of its aims was to eliminate this sort of problematic behavior. COM objects support lifetime encapsulation (i.e., a COM object is responsible for releasing itself from memory when it has no outstanding interfaces in use). All of our problems can be solved if our actor enforces the strict rule that the context data pointer of a callback key must be a pointer to a COM interface. In fact, it does not even have to be a COM object, just as long as its interface is derived from IUnknown. As we know, IUnknown is the interface from which all COM interfaces are derived and it exposes the three core methods: AddRef, Release, and QueryInterface. If our actor knows that the context data pointer for each callback key is a pointer to an IUnknown derived object, it knows that the object must also expose the Release method. Therefore, our actor need not concern itself with what type of object or structure it is pointing to, it can simply loop through each callback key in each animation set, cast the context data pointer to an IUnknown interface pointer, and

call its Release method. This will decrement the object's internal reference count. When the count reaches zero, the COM object will then unload *itself* from memory.

To be clear, we do not need to write a fully functional COM object for this system to work. Rather, we just have to make sure that the classes we allocate and assign as context data are derived from IUnknown and implement the three methods of the IUnknown interface. This is actually a useful thing for us to do because it will also go a long way towards educating you in the construction of a real COM object. We will create a class that is derived from IUnknown and behaves like a COM object in many respects. However, it will not be registered with the Windows registry like a real installed COM component; it will just be a class which we allocate using the new operator. However, this object will maintain its own reference count and delete itself from memory when this reference count reaches zero.

For this task, we will develop the CActionData class, but you can use any class you like just so long as it is derived from IUnknown and fully implements the three core COM methods.

Source Code Walkthrough – CActionData

The first class we will cover will be the CActionData class. It is a simple class and has no association with the CActor class whatsoever. In fact, this class is defined in CScene.h and will be used by our application as the class used to represent callback key context data. The class declaration is shown below:

```
class CActionData : public IUnknown
{
public:
    // Public Enumerators for This Class
    enum Action { ACTION_NONE = 0, ACTION_PLAYSOUND = 1 };

    // Constructors & Destructors for This Class.
             CActionData( );
    virtual ~CActionData( );

    // Public Functions for This Class
    // IUnknown
    STDMETHOD(QueryInterface)(THIS_ REFIID iid, LPVOID *ppv);
    STDMETHOD_(ULONG, AddRef)(THIS);
    STDMETHOD_(ULONG, Release)(THIS);

    // Public Variables for This Class
    Action  m_Action;       // The action to perform

    union                   // Some common data types for us to use
    {
        LPVOID  m_pData;
        LPTSTR  m_pString;
        ULONG   m_nValue;
        float   m_fValue;
    };
```

```
private:
    //-----------------------------------------------------------------------
    // Private Variables for This Class
    //-----------------------------------------------------------------------
    ULONG   m_nRefCount;    // Reference counter
};
```

There are a few very important points that must be made about this class. Firstly, it is derived from IUnknown, which means it can be treated like a normal COM object by the actor when it wishes to release it from memory. As far as the actor is concerned, it will see every valid callback key context data pointer as being a pointer to an IUnknown COM object and will simply call its Release method. Obviously, for this to work, this class must implement the QueryInterface, AddRef and Release methods set out in the base interface. Notice at the bottom of the declaration there is a private member variable called m_nRefCount. This will be used to represent the reference count of the object which will be incremented and decremented with calls to its AddRef and Release methods respectively. This will be set to 1 when the object is first created. The Release method will have additional work to do if it decrements the reference count of the object to 0. When this is the case, it means no code anywhere wishes to use this object any more, and it will delete itself from memory. We will look at this function in a moment.

So how does this class work? It is rather simple and can be easily extended to represents all sorts of callback key data types. As you can see, the class has defined a union so that at any one time it can store a void pointer, a string, an unsigned long, or a float. That covers virtually all bases when we consider that we could use the void pointer to point to arbitrary structures. As this is a union, all members share the same memory, so only one can be active at any one time. Which one is active depends on the setting of the m_Action variable, which can be set to a member of the Action enumerated type. Currently, this has only two members: ACTION_NONE and ACTION_PLAYSOUND. You can add different modes to this class and the code to properly release the memory for them in its destructor. For example, in Lab Project 10.1 we use the ACTION_PLAYSOUND action when we register callback key context data. We know that in this mode, the CActionData object will expect the name of a wave file to be stored as a string (i.e., the m_pString member of the union will be used). Therefore, in the destructor, if we test the m_Action member variable and find that it is set to ACTION_PLAYSOUND, we know that this represents sound effect context data and we can free the memory of the string. Obviously, if you added your own types, you would know what data it used, how it was allocated, and therefore how it must be de-allocated when the object destroys itself.

Before looking at the code to this object it will be helpful to take a look at the callback registration function implemented in CScene.cpp in Lab Project 10.1. This callback function is called by the actor when the X file is loaded. We use our new CActionData class to register two sound effect callback keys with an animation set called “cutscene_01”. This should give you a good idea about how the CActionData class is used before we discuss the code.

```
ULONG CScene::CollectCallbacks(     LPVOID pContext,
                                    LPCTSTR strActorFile,
                                    LPD3DXKEYFRAMEDANIMATIONSET pAnimSet,
                                    D3DXKEY_CALLBACK pKeys[] )
{
    CActionData * pData1 = NULL, * pData2 = NULL;
```

```
    // Bail if this is not right actor or right animation set
    if ( _tcsicmp( strActorFile, _T("Data\\AnimLandscape.x") ) != 0 ) return 0;
    if ( _tcsicmp( pAnimSet->GetName(), _T("Cutscene_01") ) != 0 ) return 0;

    // Allocate new action data objects for both callback keys
    pData1 = new CActionData();
    if ( !pData1 ) return 0;
    pData2 = new CActionData();
    if ( !pData2 ) { pData1->Release(); return 0; }

    // Setup the details for both callback keys (take off sound, and flying sound)
    pData1->m_Action  = CActionData::ACTION_PLAYSOUND;
    pData1->m_pString = _tcsdup( _T("Data\\TakeOff.wav") );
    pData2->m_Action  = CActionData::ACTION_PLAYSOUND;
    pData2->m_pString = _tcsdup( _T("Data\\Flying.wav") );

    // Store the two data objects at the correct times
    pKeys[0].Time          = 0.0;
    pKeys[0].pCallbackData = (LPVOID)pData1;
    pKeys[1].Time          = 5.0 * pAnimSet->GetSourceTicksPerSecond();
    pKeys[1].pCallbackData = (LPVOID)pData2;

    // Notify that we have two keys
    return 2;
}
```

When called by an actor, this function will return zero if the actor that called it was not loaded from a file called 'AnimLandscape.x' or if the animation set passed in is not called "Cutscene_01".

If all goes well, then it will allocate two new CActionData objects because in this example we want to register two callback keys to play sound effects at some point along the animation set's timeline. We then set the details of each CActionData object. The reference count of each CActionData object will be 1 at this point (via its constructor). We set the m_Action member of both to ACTION::PLAYSOUND. For the first CActionData object we set its string pointer member variable to point to a string containing the name of the wave file that should be played when this event is triggered ("Data\\TakeOff.wav"). We do the same for the second CActionData object, only this time we set the name of the wave file to "Data\\Flying.wav". Notice that we are using the _tcsdup function which duplicates the string passed in. This means that the string data must be freed when the CActionData object is deleted in its destructor.

Once we have the context data stored in the CActionData objects, we assign them to the context data pointer for the first two keys in the passed D3DXKEY_CALLBACK array. The first event is triggered 0 seconds into the animation set's period and the second event is triggered at 5 seconds. Finally, we return the count of two so that the actor knows that it must clone this animation set into a new one which has room for two callback keys.

> **Note**: At 0 seconds animation set time, the controller will call a callback function (more on this in a moment) which it uses to handle callback keys. This callback function will be passed the CActionData object associated with that key. The callback function can then extract the string "Data\\TakeOff.wav" from the passed object which informs it of the sound to play. At 5 seconds, this same callback function will be called again, only this time, when the string is extracted from the CActionData object, it contains "Data\\Flying.wav" causing a different sound to be played. We will discuss how the callback function should

be written later. At the moment, we are just focusing on a secure way to allocate and store the callback key context data inside the callback registration function.

You will see later that contrary to how we usually do things, when the CActionData object pointers are returned to the actor, the actor does not increment their reference count. This is because the callback function should really be seen as an extension of the actor itself, as it is called only by the actor during the loading process. Therefore, we can think of this function as allocating the object on behalf of the actor before storing its pointer in an actor-owned data structure (the callback key array). This is because we wish the reference count of each CActionData object to be exactly 1, so that when the actor releases the controller, we can fetch each callback key of each animation set and then cast its context data pointer to IUnknown. We can then call IUnknown::Release which will cause the reference count of the CActionData object to drop to 0 resulting in its unloading itself from memory. If the actor also incremented the reference count when the function returned, the CActionData objects would not destroy themselves on actor destruction because their reference counts would fall from 2 to 1 instead of from 1 to 0.

CActionData::CActionData()

The constructor sets the data area to NULL and the action of the object to ACTION_NONE. This is the default action of a newly created object which basically states that the object has no valid data which needs to be released on destruction. We saw in the above code that the callback function assigned the action to something meaningful (ACTION_PLAYSOUND) as soon as the object was allocated.

It is very important to note that we set the internal reference count of the object to 1. We wish this class to behave like a real COM object so we know that if the constructor has been called, then an instance of this class has been allocated and there will a single pointer to the object currently in use by the application.

```
CActionData::CActionData( )
{
    // Setup default values
    m_Action = ACTION_NONE;
    m_pData  = NULL;

    // ***********************************
    // *** VERY IMPORTANT
    // ***********************************
    // Set our initial reference count to 1.
    m_nRefCount = 1;

}
```

CActionData::AddRef

Just like all COM objects, our object should implement an AddRef method which increments the reference count of the object. An application should call this method whenever it makes a copy of a pointer to an object of this type. We never make any additional copies of our CActionData object in our

lab project since the only pointer to the object is stored in the D3DXKEY_CALLBACK structure. However, if your application does intend to store a copy of a pointer to this object somewhere, it should call AddRef after it has done so. This is how the object knows how many interfaces are in use by external code. We certainly would not want the object to unload itself from memory while an external code module still thought it could use its pointer to this object.

```
ULONG CActionData::AddRef( )
{
    return m_nRefCount++;
}
```

CActionData::Release

For any external code or objects that are using our object, the Release method is its way of letting us know that it is no longer interested in using it. We are used to calling the Release method of COM interfaces by now so this concept is nothing new. What is new however is the implementation of a Release method which must decrement the internal reference count and instruct the object to delete itself if the reference count reaches zero. It should now be clear why using an interface to a COM object that has been released will cause your code to crash. After the reference count reaches zero, the object and all its data simply do not exist any more.

```
ULONG CActionData::Release( )
{
    // Decrement ref count
    m_nRefCount--;

    // If the reference count has got down to 0, delete us
    if ( m_nRefCount == 0 )
    {
        // Delete us (cast just to be safe, so that any non virtual destructor
        // is called correctly)
        delete (CActionData*)this;

        // WE MUST NOT REFERENCE ANY MEMBER VARIABLES FROM THIS POINT ON!!!
        // For this reason, we must simply return 0 here, rather than dropping
        // out of the if statement, since m_nRefCount references memory which
        // has been released (and would cause a protection fault).
        return 0;

    } // End if ref count == 0

    // Return the reference count
    return m_nRefCount;
}
```

Studying the above function code you will note that the first thing the function does is decrement its internal reference count by one. If the reference count is still greater than zero then nothing much happens; the function simply returns the new reference count of the object. However, if the reference count reaches zero, then the object deletes itself from memory using the **this** pointer.

It is perfectly valid for an object to delete itself as long as you do not try to access any of the member variables from that point on. When delete is called to release the object, the destructor of the object will be called and therefore all member variables will have been released from memory. In the above code, you can see that we are very careful that we do not simply execute the last line of the function in the delete case. That is because the last line returns the value of a member variable which would no longer exist at this point and would thus cause a general protection fault. Therefore, inside the deletion case code block we return a value of zero for the reference count.

So we can see that the application should never explicitly delete an instance of this class but should instead treat it like a proper COM object using AddRef and Release semantics.

CActionData::~CActionData

The destructor of a class is the last method to be called. With our object, it will only be called once the Release method has issued the 'delete this' instruction in response to its reference count hitting zero. This is where the various modes of our CActionData objects can clean up their memory allocations. In our current object, there is only one real mode ACTION_PLAYSOUND. The object assumes that the string containing the wave filename was allocated using the _tcsdup function and as such will free the memory for this string as its last order of business.

```
CActionData::~CActionData( )
{
    // Release any memory
    switch ( m_Action )
    {
        case ACTION_PLAYSOUND:
            // Release duplicated string
            if ( m_pString ) free( m_pString ); // Allocated by '_tcsdup';
            break;

    } // End Action Switch
}
```

CActionData::QueryInterface

This function is quite an exciting one for us to look at. We have discussed how COM objects always expose this method as a means for checking the supported interfaces of an object. Indeed our object will expose two interfaces: the IUnknown interface (which all COM objects must expose) and the CActionData interface (our derived class). The CActionData interface will be used by the callback key registration callback function to assign context data to its member variables (such as the wave filenames in our example). The IUnknown interface will be used by the actor when it needs to release the callback data and has no idea of what type the context data actually points to.

As you will see, much of the mysteries of this function are removed when we look at how it is implemented in this simple example. Really, it is nothing more than a function that safely casts a pointer to the object (its 'this' pointer) to an interface/class of a specific supported type. Of course, a more

complex COM object might contain multiple classes such that some of the interfaces it exposes may work internally with completely separate objects, but in our case, there is only one object (a CActionData object). Therefore, this function is only concerned with returning one of two supported interfaces that allow the application to call the methods supported by this object.

As we have seen in the past, the QueryInterface method of IUnknown expects to be passed two parameters. The first is the alias of the GUID. As we discussed in Chapter Two, every interface has a GUID -- a unique number that identifies it. We also discussed how each GUID will often also have an alias to make it easier to remember and use. These aliases are usually in the form of "IID_I***name***", where *name* is the name of the interface. For example, if we wish to enquire about an object's support for the ID3DXAnimationSet interface, we can simply specify that interface using the IID_ID3DXAnimationSet alias instead of using its harder to remember GUID. If we look in the D3DXAnim9.h header file for example, we will see that the GUID for this interface is assigned to the alias 'IID_ID3DXAnimationSet'. The header file or the manual that ships with a software component will usually be the place to obtain a list of all aliases/interface GUIDS supported by that installed component.

Because, we have added our own interface to the mix (CActionData), we have given this interface a unique GUID and have assigned it the alias IID_CActionData. If you look in CScene.h, you will see this alias defined as follows:

```
GUID IID_CActionData = {0x736FE02A, 0x717D, 0x491F, {0x87, 0xED, 0x4B, 0x16, 0x2, 0xC8, 0xF3, 0xCA}};
```

As you can see, IID_CActionData is a variable of type GUID. The values assigned to it (i.e., the numbers in the curly braces) are the GUID itself. You can generate unique GUIDS for your own classes/COM interfaces using the GuidGen.exe tool that ships with Microsoft Visual C++. The GUIDs generated are guaranteed to be unique with respect to the GUIDs of all other interfaces in existence, so our function can safely assume that when this GUID is passed, the caller is definitely asking whether the object supports our CActionData interface, which it does.

The second parameter passed into the function should be the address of a void pointer which on function return will contain the address of the requested interface, or NULL if the GUID passed in was for an interface type that the object does not support. The application can then cast the void pointer to the correct type.

As you can see by looking at the code below, it really is just a list of conditional statements that test the past GUID to see if it is of any of the types supported by the object. If it is, we cast the ***this*** pointer, a pointer to the C++ object itself, to the desired base class before casting again to the void pointer which is ultimately returned to the function.

```
HRESULT CActionData::QueryInterface( REFIID iid, LPVOID *ppv )
{
    // Double cast so compiler can decide
    // whether it needs to switch to an alternate VTable
    if ( iid == IID_IUnknown )

        *ppv = (LPVOID)((IUnknown*)this);
```

```
    else if ( iid == IID_CActionData )

        *ppv = (LPVOID)((CActionData*)this);

    else
    {
        // The specified interface is not supported!
        *ppv = NULL;
        return E_NOINTERFACE;

    } // End iid switch

    // Call addref since a pointer is being copied
    AddRef();

    // We're supported!
    return S_OK;
}
```

When this function returns, the caller will be passed a pointer (via parameter two) to the supported interface that they requested. Notice that this function must increment the reference count (AddRef) before the function returns, since we have issued a new interface pointer to the object and an additional pointer to this object is now in existence.

The only thing that may seem confusing about the above code is how we cast the object's pointer to the requested interface/base class type before casting it to a void pointer. As we are simply casting the object to a base class, why not just cast 'this' straight to a void pointer? In this particular situation we could do that, however the double cast ensures that our function will work correctly even if any of the interfaces are derived from multiple base classes (multiple inheritance). So we should practice this safe cast policy regardless. If for example, the CActionData interface was later revised such that it had more than one base class (IUnknown and some other base class), this would cause problems as the complier would have no idea which vtable to return to the caller when the object was cast straight to a void pointer. When a class is derived from multiple base classes, it actually has multiple vtable pointers stored in the member area of each instance. By casting the 'this' pointer to the correct interface type first, *before* casting it to void, we are allowing the compiler to determine which vtable should be used for accessing functions via this interface.

> **Note:** If you would like to understand more about implementing COM objects from C++ objects, you should check out the following excellent free tutorial. It takes you step by step through the process of turning a C++ object into a COM object. It discusses the topic of multiple inheritance and multiple vtables also. This tutorial is called 'From CPP to COM' and can be found at:
>
> http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dncomg/html/msdn_cpptocom.asp

We have now implemented the three methods of IUnknown so that our object will obey the lifetime encapsulation rules set out by the COM specification. It also (as dictated by the COM standard) exposes a QueryInterface method that allows an application to ask our object if a particular interface is supported and have returned a pointer to that interface if it is. Our simple QueryInterface implementation is really just a function that allows the application to safely ask the question, "Can I cast this object pointer to a specific class type?"

We have now covered the first new class of this project (CActionData). In summary, it is a support class that allows us to store arbitrary data in an object that is capable of handling its own destruction. As discussed and demonstrated, we use an instance of this class to store the context data for each callback key that we register. In Lab Project 10.1, each callback key is used to play a sound, so the filename of the wave file is stored in CActionData object and assigned to the context data pointer of the D3DXKEY_CALLBACK structure contained in the actor's animation set. When the actor is being destroyed and the animation set is released, the actor will fetch every callback key and cast its context data pointer from a void pointer to an IUnknown interface. The actor can then call IUnknown::Release on this pointer, causing the CActionData object to destroy itself. The actor has no knowledge of what this pointer actually points to; only that it supports the Release method and will handle its own destruction. Therefore, the only rule that our actor class places on the callback system is that the context data pointer for a callback key must point to an IUnknown derived structure.

All of these pieces will fit together when we finally cover the CActor code. It was mentioned earlier that for the most part, the new methods we add to CActor should really only be lite wrappers around the underlying methods of the ID3DXAnimationController. After all, the ID3DXAnimationController handles most of the heavy lifting. We just have to assign animation sets to tracks on the mixer and play them back, right? Well, not quite! Things *should* be that simple, but unfortunately we will have to do some work just to get the animation system to work at all. This is due to a very nasty bug in ID3DXKeyframedAnimationSet.

Error in ID3DXKeyFramedAnimationSet::GetSRT

In the summer of 2003, the DirectX 9.0b SDK was released with a fully overhauled animation system. It is this new system that we have covered in this course. The changes to the animation system from previous versions were significant enough that code that had been written to work with previous versions would no longer compile.

The animation system was streamlined by the moving of animation data (keyframes) into the animation set's themselves. This had not previously been the case, as the animation data used to be stored in separate 'interpolator' objects which then had to be registered with an animation set. It is likely that this additional layer was disposed of to make the accessing of animation data by the animation set more efficient.

Previously, no AdvanceTime function existed either; in its place was a function called SetTime. Instead of passing in an elapsed time to this function, you had to pass in an actual global time which would in turn set the global time of the controller explicitly. This placed the burden on the application to keep track of the current global time of the animation controller by summing the elapsed time between frame updates. This burden has now been removed and the application now simply passes the elapsed time into the AdvanceTime call, allowing the controller to keep track of global time on our behalf.

In the 2003 summer update, the callback system was also added to the animation system. This is another worthy addition to the animation system, exposing better communication possibilities between an application and a currently playing animation.

While this animation system re-write undoubtedly caused some headaches for developers that had already coded their own animations using the previous system, the new animation system was arguably the better for the overhaul. However, in the re-write of the animation system, an error crept into the ID3DXKeyframedAnimationSet::GetSRT method, making the ID3DXKeyframedAnimationSet object essentially useless for serious development. This error seems to have gone unnoticed by the developers (and the beta testers) and as of the SDK summer update of 2004 (DirectX 9.0c) released a whole year later, the bug was still there.

The reason the bug has probably gone unnoticed for so long is that it does not always make itself apparent. For simple animated X files which play at a medium speed with basic animations, the GetSRT function seems to handle these cases fine and generates the correct SRT data for the periodic position of an animation. One such X file that seems to play without any problems using the new animation system is, ironically, Microsoft's tiny.x (ships with the SDK). This is probably the file that everyone uses to test the animation system, which might explain the oversight. Only when we began to use more complex X files did we discover the bug here in the Game Institute labs.

Indeed, the bug is actually quite hard to see if you are not paying very close attention to the screen, since the animations seem to play out correctly for the most part. However, it seems that at certain periodic positions in some animations, a single GetSRT function call will generate wildly incorrect data. This seems to happen almost at random. As the following call to GetSRT will once again generate the correct SRT data, this causes small but noticeable glitches in the animation being played. Basically, for only a few milliseconds, objects in your scene will skip or jump to completely incorrect positions in the scene as their matrices are built from bogus SRT data. A few milliseconds later, the objects are back in their correct position until the next time the GetSRT function generates invalid data. If you play the animation set from Lab Project 10.1 using the vanilla D3DX animation system, you will see these glitches as the animation plays out.

So what are the implications for the D3DX Animation System?

Since D3DXLoadMeshHierarchyFromX will always load AnimationSet objects defined in X files into ID3DXKeyframedAnimationSets, you might assume that the entire D3DX animation system that we have spent time learning about is now useless. But this is certainly not the case. We discussed in the textbook that both ID3DXKeyframedAnimationSet and D3DXCompressedAnimationSet are derived from the base interface ID3DXAnimationSet. We also learned that how we have the ability to derive our own classes from ID3DXAnimationSet so that we can create our own custom animation sets and plug them into the D3DX animation system. Provided our derived class implements the functions of the base class (GetSRT, GetPeriodicPosition, etc.) the animation controller will work fine with our derived classes. So in this workbook, we are going to write our own keyframed animation set and use that instead. Essentially we will replace all ID3DXKeyframedAnimationSets (created and registered with the controller when we load an X file), with our own CAnimationSet objects after the X file has been loaded.

On the bright side, working around this bug presents us with an opportunity to really get under the hood of the animation set object and learn how the keyframe interpolation and callback key searching is performed. Remember, it is the animation set that is responsible for generating the SRT data from its

keyframes and for returning the next scheduled callback key to the controller. We will have to implement all of this in our own animation set class, which will give us quite a bit of new insight.

> **Note:** Hopefully, by the time you read this, Microsoft will have fixed this bug, making our workaround unnecessary. However, this section will teach you how animation sets work as well as go a long way towards explaining how to create your own COM objects. It is therefore recommended that you read this section, even if the bug no longer exists. It will be helpful if you need to expand the D3DX animation system (or even write your own system from scratch) for your own reasons down the road.

Although we will not implement our animation set class as a true COM object (e.g., it will not be registered with the Windows registry), it will be close enough to the real thing. As such, we will need to implement the three methods from IUnknown (AddRef, Release, and QueryInterface), just like we did with our CActionData class.

Replacing the ID3DXKeyframedAnimationSets that have been loaded and registered with the animation controller (by D3DXLoadMeshHierarchyFromX) with our own CAnimationSet objects will be a trivial task. The CActor will have a function called ApplyCustomSets which will only be called by the actor after the X file has been loaded. The task of this function will be to copy the animation data (i.e., the keyframes) from each registered ID3DXKeyframedAnimationSet into a new CAnimationSet object. Once we have copied all the keyframed animation sets into their CAnimationSet counterparts, we will un-register each original keyframed animation set from the controller and release it from memory. We will then register our newly created CAnimationSet copies with the controller to take their place. The CAnimationSets will contain the same keyframe data and callback keys as the original ID3DXKeyframedAnimationSets from which they were copied. What we have essentially done is *clone* the animation information from each keyframed animation set into our own CAnimationSet objects. At the end of this process, our animation controller will now have *n* CAnimationSets registered with it instead of *n* ID3DXKeyframedAnimationSets. We will see how this works shortly.

The next class we will write in this lab project will be the CAnimationSet since it will be used by the code we add to CActor. It should also still be fresh in your mind after reading the textbook exactly what interaction between the animation controller and the animation set takes place. Certainly our class will need to answer requests from the animation controller in the proper way. It will need to be able to return the SRT data for a given periodic position (when the controller calls GetSRT for one of its animations) and it will also need to be able to search for the next scheduled callback key when one is requested by the controller, making sure that any passed flags (such as the one to ignore the initial position in the search) are correctly adhered to.

Source Code Walkthrough - CAnimationSet

The class declaration for CAnimationSet (see below) can be found in CActor.h and its implementation is in CActor.cpp. Notice how we derive our class from the ID3DXAnimationSet abstract base class. We did a similar thing in the last chapter when we derived our own class from ID3DXAllocateHierarchy. By doing this, we can plug our animation set object into the D3DX animation system and the controller will treat it as an ID3DXAnimationSet interface. Looking at the member functions, we can see that we must implement the three IUnknown methods (QueryInterface, AddRef, and Release) so that our object

follows the required lifetime encapsulation for COM compliance. We must also implement all of the methods from the ID3DXAnimationSet base interface because the animation controller will expect to them be implemented when asking our animation set for SRT data and callback key information.

```
class CAnimationSet: public ID3DXAnimationSet
{
  public:
     CAnimationSet( ID3DXAnimationSet * pAnimSet );
    ~CAnimationSet( );

    // IUnknown
    STDMETHOD(QueryInterface)(THIS_ REFIID iid, LPVOID *ppv);
    STDMETHOD_(ULONG, AddRef)(THIS);
    STDMETHOD_(ULONG, Release)(THIS);

    // Name
    STDMETHOD_(LPCSTR, GetName)(THIS);

    // Period
    STDMETHOD_(DOUBLE, GetPeriod)(THIS);
    STDMETHOD_(DOUBLE, GetPeriodicPosition)(THIS_ DOUBLE Position);

    // Animation names
    STDMETHOD_(UINT, GetNumAnimations)(THIS);
    STDMETHOD(GetAnimationNameByIndex)(THIS_ UINT Index, LPCTSTR *ppName);
    STDMETHOD(GetAnimationIndexByName)(THIS_ LPCTSTR pName, UINT *pIndex);

    // SRT
    STDMETHOD(GetSRT)(THIS_
                      DOUBLE PeriodicPosition,   // Position in animation
                      UINT Animation,            // Animation index
                      D3DXVECTOR3 *pScale,       // Returns the scale
                      D3DXQUATERNION *pRotation, // Returns the rotation
                      D3DXVECTOR3 *pTranslation);// Returns the translation

    // Callbacks
    STDMETHOD(GetCallback)(THIS_
                       DOUBLE Position, // Position to start callback search
                       DWORD Flags,               // Callback search flags
                       DOUBLE *pCallbackPosition, // Next callback position
                       LPVOID *ppCallbackData);   // Returns the callback data

  private:
    ULONG               m_nRefCount;      // Reference counter
    LPTSTR              m_strName;        // The name of this animation set
    CAnimationItem     *m_pAnimations;    // An Array of animations
    ULONG               m_nAnimCount;     // Number of animation items.
    D3DXKEY_CALLBACK   *m_pCallbacks;     // List of all callbacks stored
    ULONG               m_nCallbackCount; // Number of callbacks in the above list
    double              m_fLength;        // Total length of this animation set
    double              m_fTicksPerSecond;// The number of source ticks per second
};
```

Let us first examine the class member variables of this class since they will provide a better insight into how we will implement our functionality.

ULONG m_nRefCount
This member will contain the object reference count. When the object is first created, this value will be set to 1 in its constructor. If the application makes a copy of a pointer to the CAnimationSet it should call the CAnimationSet::AddRef method to force the reference count to be incremented. This will make sure that the reference count of our object is always be equal to the number of pointer variables currently pointing to it that have not yet had Release called. The Release method of this class will simply decrease the reference count and cause the object to delete itself from memory if the reference count reaches zero.

When we register the animation set with the D3DX animation controller, the controller will also call the AddRef function of our animation set object and increase the reference count. This is because, until the animation set is un-registered from the controller or the controller is released, the reference count of our object should reflect that the controller has stored a copy of the pointer. Even if our application has called Release on all of its outstanding pointers to a given animation set, the animation set would not be released from memory until the controller no longer needed access to it. In this example, only when the controller was destroyed or the animation set was un-registered, would the controller call the Release method (causing our reference count to hit zero and the object to be deleted from memory).

LPTSTR m_strName
This member is a string that will contain the name of the animation set. In our constructor, this will be extracted from the ID3DXKeyframedAnimationSet that our object is going to replace.

D3DXKEY_CALLBACK *m_pCallbacks
This variable will contain any callback keys that existed in the original ID3DXKeyframedAnimationSet passed into the constructor. The constructor will fetch all the callback keys from the original animation set and allocate this array to store them.

ULONG m_nCallbackCount
This member contains the number of callback keys in the m_pCallbacks array. If this value is set to zero then the animation set has no callback keys defined.

double m_fLength
This member will be set in the constructor to the value of the period of the original ID3DXKeyframedAnimationSet that is being copied. Therefore, this value contains the length of the animation set. We know that this is also the length of the longest running animation in the set (i.e., the highest value timestamp of all keyframes in all animations).

It is this value that will be returned to the animation controller when it calls our CAnimationSet::GetPeriod method. Our animation set also uses this value to calculate a periodic position. If looping is enabled for this animation set, then the periodic position will always be the product of an fmod between the position of the track (passed in by the controller) and the period (length) of the animation set.

double m_fTicksPerSecond
As discussed in the textbook, keyframes do not define their timestamps in seconds but rather using an arbitrary timing system called *ticks*. As our animation set's GetSRT method will always be passed a time value in seconds by the controller, it must be able to convert the seconds value into ticks so that it

can be used to find the two bounding keyframes in each of the S, R and T arrays. This value, which will be copied over in the constructor from the original keyframed animation set, specifies how many of these ticks represent a second. We can convert a value in seconds into a tick value by multiplying by m_fTicksPerSecond.

Callback keys are also defined in ticks, so this member will also be needed to convert between seconds and ticks in the GetCallback method of our class. You will recall that this function is called by the controller to locate the next scheduled callback event. In that case, the controller passes in a current position (in seconds) to begin the search. Using this value, we can convert the initial position and the callback key timestamps into the same timing system for the comparison. GetCallback will return the timestamp of the next scheduled callback key to the controller in seconds not in ticks.

CAnimationItem *m_pAnimations

In the textbook we learned that each animation set contains a number of animations and that an animation is really just a container of keyframe data for a single frame in the hierarchy. If an animation set animates five frames in a hierarchy, it will have five animations defined for it. The name of the frame in the hierarchy will match the name of the animation in the animation set that contains the keyframe data to animate it. This is how the animation controller recognizes the the pairing of hierarchy frames and animations when the set is registered with the controller.

In our implementation of CAnimationSet, we have decided to represent each animation contained in the set as a CAnimationItem object. This is another new class we will introduce in this project. If the animation set has ten animations, then this pointer will point to an array of ten CAnimationItem objects. We will cover the code to the CAnimationItem object after we have discussed CAnimationSet. We can think of a single CAnimationItem object as containing the data of a single Animation data object in an X file.

So CAnimationSet is essentially a CAnimationItem manger. The CAnimationItem object will store the keyframes (SRT Arrays) for a single frame in the hierarchy and be responsible for the keyframe interpolation (via its GetSRT method). The CAnimationSet::GetSRT function will really just act as a wrapper. Its job will be to locate the CAnimationItem that the controller has requested SRT data for, and call its CAnimationItem::GetSRT function. This function will return the SRT data back to the animation set which will in turn return it back to the controller.

ULONG m_nAnimCount

This final member stores the number of animations in the animation set (i.e., the number of elements in the array pointed to by m_pAnimations).

CAnimationSet::CAnimationSet

The constructor takes a single parameter, a pointer to an ID3DXAnimationSet interface. It is this animation set that will have its data extracted and copied by the constructor. We have previously discussed that our CActor will be calling this constructor and will pass in an ID3DXKeyframedAnimationSet so that we can generate a replacement CAnimationSet object which corrects the bug in the GetSRT method. Therefore, this constructor will need to extract the keyframe

data from the passed animation set so that it can make its own copy. However, the base ID3DXAnimationSet interface does not support functions for retrieving the keyframe data so we will first need to query the passed object for an ID3DXKeyframedAnimationSet interface. If the function succeeds then we will have an ID3DXKeyframedAnimationSet interface from which we can extract the data. If the QueryInterface method fails, then the interface we have been passed is attached to an unsupported object which does not expose the ID3DXKeyframedAnimationSet interface. If this is the case, we throw an exception.

```
CAnimationSet::CAnimationSet( ID3DXAnimationSet * pAnimSet )
{
    ULONG   i;
    HRESULT hRet;
    ID3DXKeyframedAnimationSet * pKeySet = NULL;

    // Query to ensure we have a keyframed animation set
    hRet = pAnimSet->QueryInterface( IID_ID3DXKeyframedAnimationSet,
                                     (LPVOID*)&pKeySet );
    if ( FAILED(hRet) || !pKeySet ) throw hRet;
```

If we reach this point, then it means we have a valid interface to a keyframed animation set. The first thing we do is call the ID3DXKeyframedAnimationSet::GetName method to get the name of the keyframed animation set that we are copying. If the animation set has a name, then we use the _tcsdup string duplication function to make a copy of the string and store it in the_strName member variable. We also use the methods of ID3DXKeyframedAnimationSet to copy the period of the animation set and the ticks per second value into member variables of the of the CAnimationSet object.

```
    // Duplicate the name if available
    LPCTSTR strName = pKeySet->GetName( );
    m_strName = (strName) ? _tcsdup( strName ) : NULL;

    // Store total length and ticks per second
    m_fLength           = pKeySet->GetPeriod();
    m_fTicksPerSecond   = pKeySet->GetSourceTicksPerSecond( );
```

Our next task will be to copy over any callback keys that may be contained in the set we are copying. First we call the ID3DXKeyframedAnimationSet::GetNumCallbackKeys and, provided this is greater than zero, allocate the CAnimationSet's callback key array to store this many elements. We then call the ID3DXKeyframedAnimationSet::GetCallbackKeys method and pass in a pointer to this member array. This function will copy all the callback keys in the original animation set into our object's callback keys array.

```
    // Copy callback keys
    m_pCallbacks     = NULL;
    m_nCallbackCount = pKeySet->GetNumCallbackKeys( );
    if ( m_nCallbackCount > 0 )
    {
        // Allocate memory to hold the callback keys
        m_pCallbacks = new D3DXKEY_CALLBACK[ m_nCallbackCount ];
        if ( !m_pCallbacks ) throw E_OUTOFMEMORY;
```

```
        // Copy the callback keys over
        hRet = pKeySet->GetCallbackKeys( m_pCallbacks );
        if ( FAILED(hRet) ) throw hRet;

    } // End if any callback keys
```

The next task will be to find out how many animations exist inside the passed animation set so that we can allocate that many CAnimationItem objects. Remember, a CAnimationItem represents the data for a single animation. Therefore we call the ID3DXKeyframedAnimationSet::GetNumAnimations method and use the returned value to allocate the object's CAnimationItem array to hold this many elements. We then loop through each newly created CAnimationItem in the array and call its BuildItem item method. This method will be discussed when we cover the CAnimationItem code a little later, but as you can see in the following code snippet, we pass in the ID3DXKeyframedAnimationSet interface and the index of the animation we wish to have copied into our CAnimationItem. The CAnimationItem::BuildItem method is a simple function that will fetch the SRT keys for the passed animation and make a copy of them for internal storage. After a call to BuildItem, a CAnimationItem object will contain all the keyframes that existed in the passed animation set for the specified animation index. For example, if we call:

```
m_pAnimations[5].BuildItem( pKeySet , 5 )
```

then the sixth CAnimationItem in our array will contain the SRT keyframes from the sixth animation in the keyframed animation set (pKeySet).

Here we see the code that allocates our CAnimationItem array and populates each item with data from the source animation set:

```
    // Allocate memory for animation items
    m_pAnimations = NULL;
    m_nAnimCount  = pKeySet->GetNumAnimations();
    if ( m_nAnimCount )
    {
        // Allocate memory to hold animation items
        m_pAnimations = new CAnimationItem[ m_nAnimCount ];
        if ( !m_pAnimations ) throw E_OUTOFMEMORY;

        // Build the animation items
        for ( i = 0; i < m_nAnimCount; ++i )
        {
            hRet = m_pAnimations[ i ].BuildItem( pKeySet, i );
            if ( FAILED(hRet) ) throw hRet;

        } // Next animation

    } // End if any animations

    // Release our keyframed friend
    pKeySet->Release();
```

In the code above, you can see that after we have copied the animation data into each of our CAnimationItems, the ID3DXKeyframedAnimationSet interface that we queried for earlier is released.

This is important because every time we call QueryInterface for a COM object, the reference count is incremented just before the requested interface is returned. If we forgot to do this step then the animation set would not be released when the CActor decided it was no longer needed. Remember, after this constructor returns program flow back to CActor, the keyframed animation set that was passed into the function will no longer be needed and it will have its interface released and unregistered from the controller. This will drop the reference count of the original keyframed animation set to zero, causing it to unload itself from memory.

Finally, because we wish our CAnimationSet object to behave like a COM object, we must remember to set the initial reference count of the object to 1 in the constructor.

```
    // ************************************
    // *** VERY IMPORTANT
    // ************************************
    // Set our initial reference count to 1.
    m_nRefCount = 1;
}
```

Once the constructor exits, our object will have been created and populated with the information that was contained in the ID3DXKeyframedAnimationSet passed in by the actor. The reference count of our object will also be 1, which is exactly as it should be. You will see later when we cover the CActor code that this new CAnimationSet will then be registered with the controller, in place of the original.

CAnimationSet::~CAnimationSet

The destructor to this class is nothing special; it just releases any internal memory being used by the animation set. This will be the two arrays that contain the callback keys and the CAnimationItems as well as the memory that was allocated to store the name of the animation set when _tcsdup was called in the constructor. Remember that _tcsdup uses a C alloc style function for its memory allocation, so we must release the memory using the free function.

```
CAnimationSet::~CAnimationSet( )
{
    // Release any memory
    if ( m_pCallbacks  ) delete []m_pCallbacks; m_pCallbacks = NULL;
    if ( m_pAnimations ) delete []m_pAnimations; m_pAnimations = NULL;
    if ( m_strName     ) free( m_strName ); m_strName = NULL;
}
```

What is unique about the destructor for this class is that the application code should never directly invoke it. So we should never ever write code like the following:

```
delete pMyAnimSet;
```

Our object is going to behave like a proper COM object and we never directly delete a COM object. Instead we call the interface's Release method which decrements the reference count of the object. When the reference count drops to zero, the object should delete itself from memory by doing:

```
delete this;
```

This is exactly what happens in our Release method (see next). The destructor is only ever invoked from within the class Release method.

CAnimationSet::Release

When the animation controller is destroyed by the actor, the controller will call Release for each of its registered animation sets. Therefore, our animation set must properly handle this call in COM fashion. This code is almost identical to the Release method of our CActionData class, which also followed the COM protocol.

Release decrements the reference count. When the count reaches zero, the object deletes itself by making the 'delete this' call. When we do a 'delete this', the object destroys itself in the standard way, and the destructor is invoked. When program flow finally returns back to the Release method, the object and all its member variables will have been destroyed. We must make sure that this method never tries to access any member variables or functions after this deletion (even the *this* pointer no longer exists). This is like being stranded in a static function which has no instance associated with it. As you can see, we simply return 0 after the delete.

```
ULONG CAnimationSet::Release( )
{
    // Decrement ref count
    m_nRefCount--;

    // If the reference count has got down to 0, delete us
    if ( m_nRefCount == 0 )
    {
        // Delete us (cast just to be safe, so that any non virtual destructor
        // is called correctly)
        delete (CAnimationSet*)this;

        // WE MUST NOT REFERENCE ANY MEMBER VARIABLES FROM THIS POINT ON!!!
        // For this reason, we must simply return 0 here, rather than dropping
        // out of the if statement, since m_nRefCount references memory which
        // has been released (and would cause a protection fault).
        return 0;

    } // End if ref count == 0

    // Return the reference count
    return m_nRefCount;
}
```

Note that if the object is not actually destroyed, we just decrement the reference count and return the updated value.

CAnimationSet::AddRef

When the actor adds a CAnimationSet object to the animation controller, the animation controller will expect this function to be implemented. This is because it will call the AddRef method when it stores a copy of the animation set's pointer. As you can see, this is a simple one line function that just increments the reference count.

```
ULONG CAnimationSet::AddRef( )
{
    return m_nRefCount++;
}
```

CAnimationSet::QueryInterface

The final IUnknown method we must implement is now familiar to us, given our coverage of the CActionData class. The QueryInterface function is passed an interface GUID and a void pointer and if the passed GUID interface is supported by this object, then on function return the passed void pointer will point to a properly cast version of the object.

If you take a look in CActor.h you will see that the GUID we generated (via GuidGen.exe) for the CAnimationSet interface is defined as follows, along with an IID_CAnimationSet alias:

```
GUID IID_CAnimationSet = {0x5B23B100, 0xE885, 0x4F63, {0xB0, 0x2E, 0x50, 0xFC, 0x99, 0xBA, 0x3, 0xDD}};
```

We will support three interfaces for this object. First, we obviously want to support the top level CAnimationSet interface since this is the actual type of the object. We will also support the casting of our object to an ID3DXAnimationSet interface. Remember, this interface is the base interface for all animation sets that are compatible with the D3DX animation controller. Regardless of the type of animation set registered with a controller (custom or otherwise), the controller will see all animation sets as being of type ID3DXAnimationSet. Therefore, every animation set we register must support this interface. This means of course that all the methods of this interface must be supported also. Our CAnimationSet is derived from ID3DXAnimationSet so we should implement all the methods of this interface in our object also. The third and final interface we must implement functions for is IUnknown. ID3DXAnimationSet is derived from this interface and since our class is derived from ID3DXAnimationSet, we must support the methods of its base class also. In short, the three interfaces we support are IUnknown, ID3DXAnimationSet, and CAnimationSet. Therefore, our QueryInterface implementation should allow the casting of the 'this' pointer to any of these three types.

```
HRESULT CAnimationSet::QueryInterface( REFIID iid, LPVOID *ppv )
{
    // We support three interfaces, the base ID3DXAnimationSet, IUnknown
     // and ourselves.
    // It's important that we cast these items, so that the compiler knows we
    // possibly need to switch to an alternate VTable
    if ( iid == IID_IUnknown )
        *ppv = (LPVOID)((IUnknown*)this);
```

```
    else if ( iid == IID_ID3DXAnimationSet )
        *ppv = (LPVOID)((ID3DXAnimationSet*)this);

    else if ( iid == IID_CAnimationSet )
        *ppv = (LPVOID)((CAnimationSet*)this);

    else
    {
            // The specified interface is not supported!
            *ppv = NULL;
            return E_NOINTERFACE;

    } // End iid switch

    // We're supported!
    AddRef();
    return S_OK;
}
```

Notice the double cast of the 'this' pointer – something we did in the CActionData::QueryInterface method previously. Remember, this aids the compiler in choosing the correct vtable pointer to return if multiple base classes are being derived from.

Having implemented the required IUnknown functions, we must now implement the methods of the other interface from which this object is derived: ID3DXAnimationSet. All such methods must be implemented since they will be used by the animation controller when registering and playing back the animation set. Remember, ID3DXAnimationSet is a pure abstract base class, so all of its functions *must* be implemented in our derived class.

CAnimationSet::GetName

This function is called by the animation controller to fetch the name of an animation set. We simply return a pointer to the string member variable containing the name of the animation set. The name of our animation set was copied from the ID3DXKeyframedAnimationSet passed into the constructor.

```
LPCTSTR CAnimationSet::GetName( )
{
    return m_strName;
}
```

CAnimationSet::GetPeriod

The GetPeriod method should return the length of the animation set (copied from the ID3DXKeyframedAnimationSet interface passed into the constructor). The period of an animation set is defined by its longest running animation. The longest animation is the animation which has the keyframe with the highest timestamp in the set. The period of the animation is the timestamp of this keyframe converted from ticks into seconds.

```
DOUBLE CAnimationSet::GetPeriod( )
{
    return m_fLength;
}
```

If the keyframe with the highest timestamp value had a timestamp of 2400 ticks, the period would be calculated as 2400 * TicksPerSecond. Since this value never changes, it is calculated once when the animation set is first created and then cached. As we are copying the information from an already created ID3DXKeyframedAnimationSet, we can simply extract this value and copy it into our member m_fLength variable. Refer back to our coverage of the constructor to see this value copied from the passed keyframed animation set.

CAnimationSet::GetPeriodicPosition

We discussed the duties of this function many times throughout the textbook since it is one of the most important in the system. GetPeriodicPosition will be called by the animation controller and is passed a position value in seconds. This position is actually the current time of the track to which the animation set is applied. As this position may be well outside the total length of the animation set, this function is responsible for returning a periodic position that represents the behavior of the animation set.

As an example, if the animation set was configured to loop, then the position should be mapped such that the track position should always be converted into the timeframe of the animation. If the period of the animation set was 20 seconds and a track position of 110 seconds was passed in, this would be mapped to a periodic position of 10 seconds (10 seconds into its 6^{th} loop).

Obviously, if the animation set was configured not to loop, then this would return a periodic position of 20 seconds, the last keyframe in the animation set. That is, after the track position passes 20 seconds, the same periodic position for all subsequent track positions would be returned and the same SRT data eventually generated. In this case, it would appear as if the animation set stopped playing after 20 seconds.

In our class we will treat all animation sets as looping animations. Therefore, to map the passed track position to a periodic position (a local animation set time in seconds) we simply perform the modulus of the passed track position and the period (length) of the entire animation set.

```
DOUBLE CAnimationSet::GetPeriodicPosition( DOUBLE Position )
{
    // To map, we'll just loop the position round
    double Pos = fmod( Position, m_fLength );
    if ( Pos < 0 ) Pos += m_fLength;
    return Pos;
}
```

Note: In the GetPeriodicPosition code we are careful to handle the case where the position may be specified as a negative value. For example, if the position was set to -12 for a 10 second animation then this should be treated as a 2 second loop around from the beginning of the animation. If the period of the animation was 10 seconds then -12 mod 10 would be -2. If we add the period of the animation to this (10 seconds) we get 8 seconds.

You are encouraged to add additional mapping modes beyond the simple looping logic we have implemented here. You could implement ping-pong mapping or return a periodic position that is clamped to the period of the animation set. You could also implement your own custom periodic mapping modes as well.

The final periodic position returned from this function is defined in seconds and is used by the animation controller to pass into the animation set's GetSRT method (once for each of its animations). The periodic position will be used to find the two keyframes from each SRT list that bound the time. Once the keyframes that bound the periodic position are found, they can be interpolated using a function of time. We will see this implementation in a moment when we examine the GetSRT method.

CAnimationSet::NumAnimations

This method is declared in the ID3DXAnimationSet interface and is used by the animation controller to determine how many animations are contained in the animation set. The controller must have this information so that it can set up the loop that iterates through every animation in an animation set. For each animation, it calls the GetSRT method to generate its SRT data for the current periodic position. Basically, the value returned from this method tells the controller how many GetSRT calls must be made to update this animation set.

```
UINT CAnimationSet::GetNumAnimations( )
{
    return m_nAnimCount;
}
```

This value was originally extracted from the keyframed animation set passed into the constructor. It describes the number of frames in the hierarchy that are being animated by this animation set.

CAnimationSet::GetAnimationNameByIndex

This method is called by the controller to fetch the name of a given animation in the animation set. It can be used to determine which frame in the hierarchy is being animated by the animation. This works because the name of the animation will always be the same as the name of the frame in the hierarchy which it animates.

```
HRESULT CAnimationSet::GetAnimationNameByIndex ( UINT Index, LPCTSTR *ppName )
{
    // Validate
    if ( !ppName || !*ppName || Index >= m_nAnimCount ) return D3DERR_INVALIDCALL;

    // Copy the selected item's name (it must always exist).
    _tcscpy( (LPTSTR)*ppName, m_pAnimations[ Index ].GetName() );

    return D3D_OK;
}
```

The function will be passed two parameters, the index of the animation in the set that is being inquired about and the address of a string pointer. The function uses the passed index to access its array of animations. (Remember, each animation in our array is actually a CAnimationItem object.) We copy the name of the animation at the specified index using the _tcscpy function, which we know will allocate memory for the string prior to copying it. We then assign the string pointer passed in to point to the name we have just duplicated. When the function returns, the name of the animation at the requested index (assuming it is a valid index) will be pointed to by the pointer passed in.

CAnimationSet::GetAnimationIndexByName

This next method (also from ID3DXAnimationSet) is used by the controller to fetch the index of the animation associated with the passed name. When an animation set is being registered, the animation controller calls the animation set's GetAnimationIndexByName function for each frame in the hierarchy. The controller expects to get back either a valid index via the pIndex output parameter and D3D_OK, or D3DERR_NOTFOUND via the function result.

This function allows the animation indices to be cached by the animation controller, removing the need to look up the index (or name) each time. It also helps the controller know which frames in the hierarchy even have animation data (those which resulted in D3DERR_NOTFOUND obviously do not having a matching CAnimationItem).

Once the animation controller caches the index of each animation in the set, it calls the animations set's AddRef method to increase its reference count. At this point the animation set is considered to be registered with the controller.

As you can probably guess, this is a function that simply loops through each of its animations, fetches the name of each animation and performs a string compare between the animation name and the string passed into the function. If a match is found, the index of the animation is copied into the integer whose address was passed in as the second parameter and the function returns success.

```
HRESULT CAnimationSet::GetAnimationIndexByName ( LPCTSTR pName, UINT *pIndex )
{
    ULONG i;

    // Validate
    if ( !pIndex || !pName ) return D3DERR_INVALIDCALL;

    // Search through our animation sets
    for ( i = 0; i < m_nAnimCount; ++i )
    {
        // Does the name match ?
        if ( _tcsicmp( pName, m_pAnimations[i].GetName() ) == 0 )
        {
            // We found it, store the index and return
            *pIndex = i;
            return D3D_OK;

        } // End if names match
```

```
    } // Next Item

    // We didn't find this item!
    return D3DERR_NOTFOUND;
}
```

If no match is found, we return D3DERR_NOTFOUND to the caller.

CAnimationSet::GetSRT

We have discussed at length the importance of the animation set's GetSRT function and the role it plays in the overall animation system. It is probably not overstating the point to say that this method is the core of the entire animation engine.

GetSRT is called by the controller for each animation in the animation set. Its job is to calculate the SRT data for the current periodic position. The periodic position is passed into the function along with the index of the animation that we wish to generate SRT data for. The method is also passed a scale vector, a rotation quaternion, and a translation vector, which on function return should be filled with the SRT data for the specified time.

While you might expect this function to be large, it is actually just a simple wrapper around the CAnimationItem::GetSRT method, which we will discuss shortly.

```
HRESULT CAnimationSet::GetSRT( DOUBLE PeriodicPosition, UINT Animation,
                               D3DXVECTOR3 *pScale, D3DXQUATERNION *pRotation,
                               D3DXVECTOR3 *pTranslation )
{
    // Validate
    if ( Animation > m_nAnimCount ) return D3DERR_INVALIDCALL;

    // Return the interpolated result
    return m_pAnimations[ Animation ].GetSRT( PeriodicPosition,
                                              pScale,
                                              pRotation,
                                              pTranslation );
}
```

As you can see, the function uses the passed animation index to access the CAnimationItem in its animation array. It then calls its GetSRT method passing in the periodic position and the three structures that will contain the SRT output. Remember, the CAnimationItem class represents the data and members for a single animation in the set. It is a keyframe container that exposes its own GetSRT method which performs the interpolation to generate the SRT data for the passed periodic position for a single animation.

If the animation set contained 50 animations, then we would expect the animation controller to call the CAnimationSet::GetSRT method 50 times in a single update. Each time, the index of a different animation would be passed. This is used to call the GetSRT method of the corresponding CAnimationItem in the array and generate its SRT data. Our function is really nothing more than a function that routes an animation request to the GetSRT method of the correct animation.

CAnimationSet::GetCallback

The final method that we need to implement in our CAnimationSet class is called GetCallback (an ID3DXAnimationSet interface method). As discussed in the textbook, this method is passed a track position by the controller and will perform a search starting from that position to locate the time and data for the next scheduled callback event in the track timeline. Remember, the controller will cache this callback information and will not need to call this method again until the callback has been executed.

The code that follows shows the first section of the function. The function is passed a time (in seconds) by the animation controller describing the track position from which to begin the search for the next callback key. The time value passed in depends on how the function is being called by the animation controller. As explained in the textbook, if the controller is having AdvanceTime called for the first time, or if the global time of the controller has been reset, the callback event cache of the controller will be in an invalid state. When this is the case, the controller will call the GetCallback method of the animation set passing in the current position of the track (i.e., the track time). It will be returned the next scheduled callback event on the track timeline at some point in the future. The animation controller will then cache this information so that it knows it does not have to perform this search again until the position of the track has passed the cached callback key timestamp causing event execution.

If the callback event cache is not invalid, then the controller will call this function passing in the previously cached event time as the position parameter. For example, imagine the first time the AdvanceTime method is called and the controller calls the animation set's GetCallback method, passing in the current track position (0 in this case). The function might return a callback key with a timestamp of 10 seconds. This information will be cached by the controller so that it knows it does not have to do anything else with the callback system until 10 seconds (track time) has been reached. Approximately 10 seconds later, when the AdvanceTime call happens again, the controller would determine that the current track position is now larger than the previously cached callback key timestamp. The callback key is executed (i.e., the callback handler is called) and now it is time to find a new 'next' scheduled callback key. The controller calls the GetCallback method of the animation set again but will not pass in the current track position (which may be larger than 10 seconds). Instead, the previously cached callback time (10 seconds) will be passed in as the position parameter. This will cause the function to begin searching for a new callback key from the position of the previous callback key.

Remember that if the above system was not employed, callback events could be skipped. For example, imagine that we have a scheduled next callback event of 10 seconds, but the next time GetCallback is called, 12 seconds have passed. The 10 second event would be triggered normally because the controller would correctly detect that the position of the track (12 seconds) has passed the currently cached callback time (10 seconds). However, if we were to then perform the search for the next scheduled callback key starting at the current track position (12 seconds), we have risked skipping an event that may have been registered at 11 seconds because we jumped straight from 10 to 12 seconds. Therefore, by always searching for the next scheduled callback event from the position of the previously cached callback key, we make sure that we catch and trigger all callback events, even if the actual track position is several seconds ahead of the callback key processing pipeline.

The second parameter to this function can be a combination of zero or more flags that control how the search is performed.

If the controller passes in the D3DXCALLBACK_SEARCH_EXCLUDING_INITIAL_POSITION flag, then our function should not return a callback key which shares the same timestamp as the initial position passed in. In the textbook we learned that the controller will always pass this flag if a previously cached callback exists. This is because the position passed in will be the time of the previous timestamp and, if we do not ignore the initial position in the search, we will end up returning that same callback key that has already been executed. The exception is when the controller has an invalid cache; in this case, no previous callback exists and the controller is interested in the very first callback key that exists from the passed initial position (including that initial position).

Another flag that can also be combined in this parameter is the D3DXCALLBACK_SEARCH_BEHIND_INITIAL_POSITION flag, which specifies that the call would like the search for the next callback key to be done in the reverse direction (i.e., searching backwards from the passed initial position to find the next 'previous' key). Interestingly, in our laboratory tests, we never seemed to be passed this flag by the D3DXAnimationController, so it is very hard to determine with accuracy when the controller might pass this flag. It may be that it never does and that this is simply a flag that the application can use for its own callback key queries. You might assume that it is used by the controller when an animation set is playing in ping-pong mode and the direction of the animation changes with every odd and even loop. However, this is highly unlikely since the GetPlayBack type method is not a member of the ID3DXAnimationSet base interface and therefore the controller would have no way of accessing whether an animation set was configured to loop or ping-pong. At the time of writing there was no exporting software supporting the ping-pong animation mode, so these are ultimately untested conjectures. So it may be that this search mode is just for an application's benefit. After all, use of the GetCallback method is not restricted to the controller; it can also be a utility function used by your application to search for callback keys. Either way, the ID3DXKeyframedAnimationSet implementation supports the backwards search mode, so our implementation will do so as well.

The final two parameters are output parameters. The controller will pass in the address of a double and the address of a void pointer and on function return, the double should contain the track time (in seconds) of the next scheduled callback key and the void pointer should point to any context data (a CActionData object in our case) that was associated with the callback key when it was registered. This context data will be passed to the callback handler function (covered later) when the callback event is executed.

Here is the first small section of the function that requires some explanation:

```
HRESULT CAnimationSet::GetCallback( DOUBLE Position, DWORD Flags,
                                    DOUBLE *pCallbackPosition,
                                    LPVOID *ppCallbackData )
{
    long                i;   // Must be signed long for the backwards search
    D3DXKEY_CALLBACK    Key;

    // Are there any keys at all?
    if ( m_nCallbackCount == 0 ) return D3DERR_NOTFOUND;

    // Modulate position back into our range (and convert to ticks)
    double fPeriodic = fmod( Position, m_fLength ) * m_fTicksPerSecond;
    double fLoopTime = (Position * m_fTicksPerSecond) - fPeriodic;
```

The first thing the code does is test to see if this animation set even has any callback keys registered. If not, the function can return immediately. Notice that we return the D3DERR_NOTFOUND result expected by the controller when no more callback keys exist on the track timeline.

The next two lines of code might seem a bit confusing, so let us examine the time conversions which must be performed in this function in order to return the next callback key to the controller as a track position.

As we know from our earlier discussions, callback keys, just like SRT keys, are defined in ticks. Along with SRT keys, they help define the period of the animation set. If an animation set has a period of 10 seconds, then we know that any callback keys will have timestamps within this period. However, the controller is not interested in getting back a callback key timestamp as a periodic position because it has no concept of animation set local time. The controller wants to know when the next callback event will occur in track time, and as we know, the two are very different.

Let us imagine that we have a looping animation set with a period of 10 seconds which contains a single callback key at 10 seconds. We know that the track position will continue to increase with calls to AdvanceTime, but the periodic position of the animation set will loop every 10 seconds. If we simply searched for a callback key from the passed track position each time, once the track position exceeded 10 seconds (and continued to increase) we would never find another. The next time the animation set looped, our sound effect (for example) would not play. Therefore, we need to map the passed track position into a periodic position, perform the search for the next callback key starting from that periodic position, and, once found, convert that periodic position back into a future track position and return it to the controller.

With our 10 second animation set example, at a track position of 15 seconds, the callback key (at 10 seconds periodic position) would have been executed once. Now the controller would call the GetCallback method again to find another callback key, passing in a track position of 15 seconds. Our function would convert this into a periodic position of 5 seconds (i.e., 5 seconds into the second loop) and it would start the search and find our callback key at 10 seconds (periodic position).

Of course, we cannot simply return this periodic position of 10 seconds back to the controller because the track position is already at 15 seconds and it wants to know the next (future) time that this callback key needs to be executed in track time. So our function, having been passed a track position of 15 seconds, first maps this to a 5 second periodic position. We calculate that value in the prior code by performing the modulus of the passed track position and the period of the animation. This gives us:

PeriodicPosition = fmod (TrackPosition , Period) * TicksPerSecond
PeriodicPosition = fmod (15 , 10)
PeriodicPosition = 5

This calculation maps the track position to a periodic position. We know that whatever loop we are on, we are currently 5 seconds through it. You can see that in the above code listing, we perform this calculation and store the result in the local variable fPeriodic. This is the periodic position from which we wish to begin our search for callback keys. Notice that when we perform that calculation, we also multiply the result by m_fTicksPerSecond so that the periodic search start position is now specified in

ticks. Since all of our keys are specified with tick timestamps, we now have our start position using the same timing type as the callback keys for the search.

You might think that we now have everything we need, but this is not the case quite yet. While the periodic position certainly allows us to search and find the next scheduled callback key, its timestamp will be a periodic position. The controller will want to get that timestamp back as a track position, so the timestamp of the callback we eventually return must be converted from a periodic position into a track position. The problem is that normally the animation set would not have any idea which iteration of a loop it is on because it does not record how many times it has looped. However, this is precisely why the animation controller passes us a track position and not a periodic position (as it does with the GetSRT method). It is only when we have the track position from which to begin the search do we have the necessary information to turn the periodic timestamp of a key back into a track position.

If you look at the previous code listing you will see that we calculate a value and store it in a local variable called fLoopTime. This uses the passed track position and the previously calculated periodic position to get the number of complete loops the animation has performed.

In our example, we said the function was passed a track position of 15 seconds and the period of the animation set was 10 seconds. At this point we have also calculated the periodic position as being 5 seconds. Therefore, if we subtract the current periodic position from the track position, we have the number of seconds of complete animation loops that have been performed:

LoopTime = TrackPosition – PeriodicPosition
LoopTime = 15 – 5
LoopTime = 10 seconds of completed loops.

This is exactly what we do in the first section of the function code shown previously -- we simply subtract the periodic position from the track position. However, our periodic position is currently in ticks, so we convert the track position to ticks also. What ends up in the fLoopTime local variable is the number of ticks of complete loops that the animation has played out. This can also be thought of as the track position at which the current animation loop began (converted to ticks).

So we now know what we have stored in the two local variables. But why do we need know the total loop time? Well, in our example we said that the callback key was at a periodic position of 10 seconds. Therefore, we would begin our search from 5 seconds into the callback key array and find that the next callback key encountered has a periodic position of 10 seconds. We know that this is a periodic position, but the controller wants this to be a track position. That conversion is now very easy. We simply add the timestamp of the callback key to the total loop time (the track position at which the current loop began) and we have the track position in the future that this key is scheduled for:

FutureTrackPos = LoopTime + TimeStamp
= 10 + 10
= 20 seconds track time this key will be triggered

As you can see, by simply adding the timestamp of the key (which is a periodic position) to the total loop time, we end up with that future periodic position as a track time. The periodic position is

essentially like an offset from the track position when this current animation loop started. And we know that this works out properly. If we have a 10 second long looping animation set which has a single callback key at 10 seconds (periodic position), we know that the callback key should be triggered at 10 seconds, 20 seconds, 30 seconds, etc. (i.e., every time the track position crosses the 10 second boundary of the animation set's period).

That was quite a long discussion to explain two lines of code, but now we know what these local variables contain and how we need to map the resulting timestamp back into a track position. The rest of the function will be easy to understand.

The remainder of the function is mostly divided into two code blocks depending on whether the `D3DXCALLBACK_SEARCH_BEHIND_INITIAL_POSITION` flag has been specified by the controller. You will recall from the textbook that this flag instructs our function to perform the search for the next callback key from the initial position in a backwards direction instead of the more common forward direction.
We will take a look at the code that deals with finding the next callback key in forward search mode first since this is the mode used by the controller all of the time. In other words, this is the code that is executed if the `D3DXCALLBACK_SEARCH_BEHIND_INITIAL_POSITION` flag is not passed by the controller.

```
    // Searching forwards?
    if ( !(Flags & D3DXCALLBACK_SEARCH_BEHIND_INITIAL_POSITION) )
    {
        // Find the first callback equal to, or greater than the periodic position
        for ( i = 0; i < m_nCallbackCount ; ++i )
        {
            if ( m_pCallbacks[i].Time >= fPeriodic ) break;
        }// Next Callback

        // If nothing was found, then this HAS to be the first key
        // (for the next loop)
        if ( i == m_nCallbackCount )
        {
            // Increase the loop time and use the first key
            fLoopTime += (m_fLength * m_fTicksPerSecond);
            i = 0;

        } // End if wrap
```

This code block sets up a loop to iterate through each callback key in the callback key array. Inside the loop, we compare the timestamp of each callback key and break when we find the first callback key that has a timestamp that is either greater than or equal to the search start time. What we are doing here is simply trying to find the first callback key that is not behind the search start time. As this is our forward search mode, we are not interested in any callback keys that are behind the track position passed in, as they will have already been executed.

Once we find a suitable callback key, we break from the for loop. At this point the loop variable 'i' will contain the index of the key in the array that we are potentially interested in. You can see in the above code that once outside the loop, if 'i' is equal to the number of callback keys, then no future callback keys exist. We know then that the search start time was greater than the timestamps of all callback keys in the array. When this is the case, we essentially need to loop around the animation. So we know that the first key in the array is now the key we want as we have executed all keys right up to the end of the

array and it is time to start again from the beginning. However, we cannot just simply set 'i' to zero and leave it at that, since we need to account for the fact that we have just performed another loop of animation that we did not account for when we calculated the fLoopTime variable at the start of the function. Since we have searched to the end of the current period and have looped back around to the beginning again, we have just performed an additional animation loop. Therefore, we add onto the fLoopTime variable (which stores the current number of animation loops in ticks) the period of the animation set (in ticks).

For example:

AnimationSet.Period =10 seconds
CallBackKey.Time = 5 seconds
StartSearchTrackPosition = 18 seconds
StartSearchPeriodicPosition = 8 seconds
LoopTime = 10 seconds

The above settings demonstrate how this function might be called for an animation set with a period of 10 seconds that has a single callback key registered at 5 seconds. The function is passed a start search time (as a track position) of 18 seconds. We showed earlier how a modulus of this value with the period of the animation set will convert the search start time into a periodic position of 8 seconds.

Now we know, just by looking at the data described above, that from a track position of 18 seconds, the next callback key should actually be at track position 25 seconds (five seconds into the third loop). So we perform the search for the next callback key from 8 to 10 seconds and we do not find one; this is because the only callback key has a timestamp less than 8 seconds. This means we need to loop. We know that the first callback key is the callback key we are interested in, and we also increment the loop time by the period of the animation set:

LoopTime + = 10
LoopTime = 20 seconds

Now that we have compensated for the additional loop, later in the function we will be able to calculate the track position of the callback key as:

TrackPos = LoopTime + CallBackKey.Time;
= 20 + 5
= 25 seconds.

And there we have it. The addition to the loop time variable in the above code lets us keep a record of the number of animation loops performed. It is used later in the function to convert the timestamp of the callback key back into a future track position.

At this point we now have the index of the callback key stored in loop variable 'i'. The next section of code (the conclusion of the forward search) fetches this key from the array and then performs several tests for compliance with the input search flags.

```
        // Get the key
        Key = m_pCallbacks[ i ];

        // Do we ignore keys that share the same position?
        bool bExclude = Flags & D3DXCALLBACK_SEARCH_EXCLUDING_INITIAL_POSITION;

        if ( (bExclude && Key.Time == fPeriodic) )
        {
            // If we're going to wrap onto the next loop, increase our loop time
            if ( (i + 1) == m_nCallbackCount )
              fLoopTime += (m_fLength * m_fTicksPerSecond);

            // Get the next key (taking care to loop round where appropriate)
            Key = m_pCallbacks[ (i + 1) % m_nCallbackCount ];

        } // End if skip to the next key

    } // End if searching FORWARDS
```

In the above code, once the callback key data is fetched from the array we test to see if the caller (the animation controller) specified `D3DXCALLBACK_SEARCH_EXCLUDING_INITIAL_POSITION`. If so, then we are not interested in this key if it has a timestamp that is equal to the search position. As you can see, if the flag is set and the key timestamp is equal to the periodic position then we need to ignore this key and fetch the next one in the array -- element i+1. Notice the conditional that states if i+1 is equal to the number of callback keys, then the original key was the last one in the callback array and we need to loop around and fetch the first callback key. If this is the case, we record this loop in the fLoopTime variable so that we always have an accurate record of exactly how many ticks of animation looping has been performed by this animation set. Finally, we fetch the actual key we need. Notice the code we use to index into the correct array element. We use the modulus of i+1 and the number of callback keys in the array. If i+1 is equal to the number of callback keys, zero is returned from the operation and the first callback key in the array is used as required.

We have now covered the code to find the correct callback key when we are in forward searching mode. The next section of code shows essentially the reverse operation being performed. We will not take this code block line by line as it is just a mirror image of the code we have just discussed. Rather than searching forward from the current search position and looping back around to the beginning when necessary, it searches backwards from the passed search position looping back around to the end of the array when necessary.

```
// else we are in backwards search mode
    else
    {
        // Find the first callback equal to, or greater than the periodic position
        for ( i = (signed)m_nCallbackCount - 1; i >= 0; --i )
        {
            if ( m_pCallbacks[i].Time <= fPeriodic ) break;

        } // Next Callback

        // If nothing was found, then this HAS to be the last key
        //(for the next loop)
        if ( i == -1 )
```

```
        {
            // Decrease the loop time and use the last key
            fLoopTime -= (m_fLength * m_fTicksPerSecond);
            i = (signed)m_nCallbackCount - 1;

        } // End if wrap

        // Get the key
        Key = m_pCallbacks[ i ];

        // Do we ignore keys that share the same position?
        bool bExclude = Flags & D3DXCALLBACK_SEARCH_EXCLUDING_INITIAL_POSITION;
        if ( (bExclude && Key.Time == fPeriodic) )
        {
            // If we're going to wrap onto the previous loop,
            // decrease our loop time
            if ( (i - 1) < 0 ) fLoopTime -= (m_fLength * m_fTicksPerSecond);

            // Get the next key (taking care to loop round where appropriate)
            Key = m_pCallbacks[ (i - 1) % (signed)m_nCallbackCount ];

        } // End if skip to the next key

    } // End if searching forwards
```

At this point in the function we have located our callback key. The controller wants us to store the track position of this callback key in the pCallbackPosition output parameter it passed in, so we assign the sum of the loop time and the timestamp of the callback key. Since both the loop time and the callback key timestamp are in ticks, we must divide the result by the TicksPerSecond ratio of the animation set so that it is a track position specified in seconds. We also assign the context data pointer of the callback key (which in our code will actually be a void pointer to a CActionData object) to the ppCallbackData pointer passed in by the contoller. The last section of the code that performs these steps is shown below.

```
    // Return the time, mapped to the track position (converted back to seconds)
    if ( pCallbackPosition )
        *pCallbackPosition = (Key.Time + fLoopTime) / m_fTicksPerSecond;

    // Return the callback data
    if ( ppCallbackData )
        *ppCallbackData = Key.pCallbackData;

    // Success!!
    return D3D_OK;
}
```

So we see that there is nothing magical about the functions that are implemented by ID3DXKeyframedAnimationSet. While we took a black-box view of them in the textbook, now we have had a chance to see exactly how the animation set passes back information to the animation controller in response to a next scheduled callback key search.

This concludes our coverage of the code and methods in our CAnimationSet class. We still have a ways to go yet though with respect to understanding the system functionality. Next we will need to look at the code that generates SRT data using interpolation. The keyframe interpolation code is tucked away in the

CAnimationItem class, so we will look at this class next to wrap up coverage of our custom animation set.

Source Code Walkthrough - CAnimationItem

For every animation in the set, the keyframe data will be stored in a CAnimationItem object. If our animation set contains 10 animations, it will have an array of 10 CAnimationItem objects. Each item in this array will contain the keyframe data for a single frame in the hierarchy and will expose the methods to build SRT data for any periodic position and return it to the animation set. The set then hands the SRT information back the animation controller.

While this class has only a few methods, it does contain one of the most important in our system -- GetSRT. As we saw in our coverage of CAnimationSet::GetSRT, that function acted as a wrapper around a call to the GetSRT method of a CAnimationItem object at the requested animation index.

Let us first look at the class declaration (see CActor.h).

```
class CAnimationItem
{
public:
    //-----------------------------------------------------------------------
    // Constructors & Destructors for This Class.
    //-----------------------------------------------------------------------
     CAnimationItem( );
    ~CAnimationItem( );

    // Public methods for the class
    LPCTSTR GetName     ( ) const;
    HRESULT GetSRT      ( DOUBLE PeriodicPosition, D3DXVECTOR3 *pScale,
                          D3DXQUATERNION *pRotation, D3DXVECTOR3 *pTranslation );
    HRESULT BuildItem   ( ID3DXKeyframedAnimationSet *pAnimSet, ULONG ItemIndex );

private:
    LPTSTR              m_strName;              // The name of this animation
    D3DXKEY_VECTOR3    *m_pScaleKeys;           // The scale keys
    D3DXKEY_VECTOR3    *m_pTranslateKeys;       // The translation keys
    D3DXKEY_QUATERNION *m_pRotateKeys;          // The rotation keys
    ULONG               m_nScaleKeyCount;       // Number of scale keys
    ULONG               m_nTranslateKeyCount;   // Number of translation keys
    ULONG               m_nRotateKeyCount;      // Number of rotation keys
    double              m_fTicksPerSecond;      // Source ticks per second

    ULONG               m_nLastScaleIndex;     // Cache for last 'start' scale key
    ULONG               m_nLastRotateIndex;    // Cache for last 'start' rot key
    ULONG               m_nLastTranslateIndex; // Cache for last 'start' trans key
    double              m_fLastPosRequest;     // Cache for last periodic position
};
```

The CAnimationItem class exposes three methods. The GetName function returns the name of the animation, which will also match the name of the frame in the hierarchy that it animates. The GetSRT method is responsible for generating the SRT data for a given periodic position. The BuildItem method is used by our CAnimationSet class constructor to copy over the keyframe data from the original animation in the ID3DXKeyframedAnimatonSet.

You will recall how the CAnimationSet constructor is passed a pointer to an ID3DXKeyframedAnimationSet which it will essentially clone. The code fetches the number of animations contained in the animation set and then uses this value to allocate a CAnimationItem array of the same size. The code then looped through each CAnimationItem and called its CAnimationItem::BuildItem function. The function is passed the interface of the ID3DXKeyframedAnimationSet and the index of the animation which we would like to copy into this CAnimationItem. As a reminder, here is a small excerpt from the CAnimationSet constructor which we discussed earlier:

```
// Allocate memory for animation items
    m_pAnimations = NULL;
    m_nAnimCount  = pKeySet->GetNumAnimations();
    if ( m_nAnimCount )
    {
        // Allocate memory to hold animation items
        m_pAnimations = new CAnimationItem[ m_nAnimCount ];
        if ( !m pAnimations ) throw E OUTOFMEMORY;

        // Build the animation items
        for ( i = 0; i < m_nAnimCount; ++i )
        {
            hRet = m_pAnimations[ i ].BuildItem( pKeySet, i );
            if ( FAILED(hRet) ) throw hRet;

        } // Next animation

    } // End if any animations
```

The BuildItem function is responsible for extracting the keyframe data from the specified animation in the ID3DXKeyframedAnimationSet and copying it into internal SRT arrays. We will see this function in a moment.

Let us now discuss the members of the CAnimationItem class.

LPTSTR m_strName

This member points to a string that contains the name of the animation. There will also be a frame in the hierarchy with a matching name. The name of the frame and the animation essentially form a link for the controller between the animation's keyframe and the frame matrix in the hierarchy it is to animate.

D3DXKEY_VECTOR3 *m_pScaleKeys

If this animation contains any scale keyframes, then this will point to an array of D3DXKEY_VECTOR3 keys. The array will be allocated in the BuildItem method when the scale key

list is copied over from the matching animation in the ID3DXKeyframedAnimationSet. If no scale keyframes are defined for this animation, this member will be set to NULL.

D3DXKEY_VECTOR3 *m_pTranslateKeys
If this animation contains any positional keyframes, then this will point to an array of D3DXKEY_VECTOR3 keys. The array will be allocated in the BuildItem method when the translation key list is copied over from the matching animation in the ID3DXKeyframedAnimationSet. If no translation keyframes are defined for this animation, this member will be set to NULL.

D3DXKEY_QUATERNION *m_pRotateKeys
If this animation contains any rotation keyframes, then this will point to an array of D3DXKEY_QUATERNION keys. The array will be allocated in the BuildItem method when the rotational keyframe list is copied over from the matching animation in the ID3DXKeyframedAnimationSet. If no scale keyframes are defined for this animation, this member will be set to NULL.

The three arrays just described contain the SRT keyframes of the animation. These are the keyframes that will be used for the interpolation and generation of SRT data in the GetSRT function.

ULONG m_nScaleKeyCount
ULONG m_nTranslateKeyCount
ULONG m_nRotateKeyCount
These three members will be set in the BuildItem method. They describe the number of scale keys, translation keys, and rotation keys stored in the m_pScaleKeys, m_pTranslateKeys and m_pRotateKeys arrays respectively.

double m_fTicksPerSecond
This member will store the TicksPerSecond ratio of its parent animation set. We will need to access this value frequently during interpolation in the GetSRT method, so we copy this value from the parent animation set and cache it for ease of access. Remember that this is a per-set property not a per animation property, so all CAnimationItems in the same CAnimationSet will have the same value.

ULONG m_nLastScaleIndex
ULONG m_nLastRotateIndex
ULONG m_nLastTranslateIndex
These three members are used to optimize the GetSRT method. They are stored so that we do not have to loop through all keyframes in all three arrays to find the bounding keyframes for the passed periodic position every time GetSRT is called by the controller. For example, for periodic position A, we might find that the two bounding scale keyframes are at indices m_pScaleKeys[25] and m_pScaleKeys[26]. What we will do at this point is store the index 25 in the m_nLastScaleIndex member. The next time GetSRT is called for this animation for periodic position B, we can assume a forward moving timeline and begin the search for bounding keyframes starting at scale key 25. This optimizes our search for keyframes by ignoring those that are behind the last cached position and reduces loop traversal overhead.

It should be noted that this is an efficient optimization only when the timeline is being advanced in a forward direction (which is almost all of the time). It provides no speed benefit when playing the animation in the reverse direction (e.g., in ping pong mode where every other loop the periodic position is stepping backwards). We must also make sure that we invalidate the cache as soon as the current periodic position wraps around and becomes smaller than the last cached key time. This is all handled in the GetSRT method, which we will look at in a moment.

double m_fLastPosRequest

This final member is used to store the periodic position of a previous call to the GetSRT method. For example, when the GetSRT method is called, the current periodic position is stored in this member. The next time the GetSRT method is called, we can test the new periodic position against the previous one. If the new periodic position is less than the previous position, we know that the animation has looped (or moved backwards) and we can set the three cached indices (m_nLastScaleKey, m_nLastRotateIndex and m_nLastTranslateIndex) to zero. This make sure that when looping occurs, the search for bounding keyframes always begins at keyframe 0 in each array the next time through.

Let us now examine the three methods exposed by this object. Ww will begin with the CAnimationItem::BuildItem method.

CAnimationItem::BuildItem

BuildItem is called by the CAnimationSet constructor for each CAnimationItem created. The job of this function is to populate the CAnimationItem's internal arrays with the SRT data from the ID3DXKeyframedAnimationSet that is in the process of being cloned and replaced. Since a CAnimationItem object represents the data for a single animation in the set, it is passed the interface to the keyframed animation set that needs to be copied and the index of the animation within that set that will be copied.

The function first tests that the index passed in is within the number of animations contained in the source keyframed animation set. It exits if this is not the case. If the animation index is valid, we use the ID3DXKeyframedAnimationSet::GetAnimationNameByIndex method to fetch the name of the animation within the animation set. As you can see in the following code, we call this method passing in the index of the animation within the set that we wish to retrieve the name for, and a pointer to a string that will point to the name of the animation on function return. We use the tcsdup method to duplicate the string so that we can store a copy of the animation name in the CAnimationItem::m_strName member variable. This name should match the name of a frame in the hierarchy and the name of a registered animation output.

```
HRESULT CAnimationItem::BuildItem( ID3DXKeyframedAnimationSet *pAnimSet,
                                   ULONG ItemIndex )
{
    LPCTSTR strName = NULL;

    // Validate
    if ( !pAnimSet || ItemIndex > pAnimSet->GetNumAnimations( ) )
        return D3DERR_INVALIDCALL;
```

```
    // Get the name and duplicate it
    pAnimSet->GetAnimationNameByIndex( ItemIndex, &strName );
    m_strName = _tcsdup( strName );
```

In the next section we retrieve the number of scale, rotation and translation keys that exist for the animation we are currently in the process of copying. We will obviously need these values to allocate the SRT arrays of the CAnimationItem and to loop through the keyframes in the GetSRT method. We also extract the TicksPerSecond ratio from the animation set and store that in a CAnimationItem member variable. This is an animation set value that will be consistent for all animations in the set, and we store a copy of it inside each CAnimationItem so that we can easily access it from GetSRT without having to query into the parent animation set.

```
    // Store any secondary values
    m_fTicksPerSecond    = pAnimSet->GetSourceTicksPerSecond( );
    m_nScaleKeyCount     = pAnimSet->GetNumScaleKeys( ItemIndex );
    m_nRotateKeyCount    = pAnimSet->GetNumRotationKeys( ItemIndex );
    m_nTranslateKeyCount = pAnimSet->GetNumTranslationKeys( ItemIndex );
```

At this point, we know how many keyframes exist for this animation in each of its scale, rotation, and translation arrays. We will need to allocate these arrays in our CAnimationItem if we intend to copy and store this data from the ID3DXKeyframedAnimationSet. In the following code you can see that we use the values just retrieved to allocate the SRT arrays:

```
    // Allocate any memory required
    if ( m_nScaleKeyCount )
        m_pScaleKeys  = new D3DXKEY_VECTOR3[ m_nScaleKeyCount ];

    if ( m_nRotateKeyCount)
        m_pRotateKeys = new D3DXKEY_QUATERNION[ m_nRotateKeyCount ];

    if ( m_nTranslateKeyCount )
        m_pTranslateKeys = new D3DXKEY_VECTOR3[ m_nTranslateKeyCount ];
```

With our arrays now allocated, all that is left to do is copy the keyframe data from the animation in the ID3DXKeyframedAnimationSet into our own SRT arrays. Luckily, ID3DXKeyframedAnimationSet provides an easy means for retrieving this data. We can retrieve the scale, rotation or translation keys simply by calling the GetScaleKeys, GetRotationKeys, and GetTranslationKeys methods. We pass these functions the index of the animation that we wish to retrieve the keyframes for (the first parameter), and a pointer to a pre-allocated array that will receive the copy of the keyframe data (the second parameter). These arrays are the ones we just allocated. The functions will then copy the keyframe data from the ID3DXKeyframedAnimationSet directly into our CAnimationItem SRT arrays.

```
    // Retrieve the keys
    if ( m_pScaleKeys     )
        pAnimSet->GetScaleKeys( ItemIndex, m_pScaleKeys );

    if ( m_pRotateKeys    )
        pAnimSet->GetRotationKeys( ItemIndex, m_pRotateKeys );
```

```
    if ( m_pTranslateKeys )
        pAnimSet->GetTranslationKeys( ItemIndex, m_pTranslateKeys );

    // We're done!
    return D3D_OK;
}
```

At this stage, our CAnimationItem contains all SRT keyframe data for a single frame in the hierarchy. When the CAnimationSet constructor has called this method for each animation, we will have copied all the animation data from the original keyframed animation set. We can then use our CAnimationSet instead of the original ID3DXKeyframedAnimationSet and sidestep the bug in the ID3DXKeyframedAnimationSet::GetSRT method.

CAnimationItem::GetName

The second method of the CAnimationItem class is a utility function which returns the name of the animation item. It is vital that this function be available to our CAnimationSet class. Recall that our CAnimationSet class must provide a function called GetAnimationNameByIndex (part of the ID3DXAnimationSet base interface). The animation controller will expect this function to be present in our animation set so that it can be called to fetch the name of an animation at the specified index. In our implementation, animations are stored in a CAnimationItem array inside the CAnimationSet. The CAnimationSet::GetAnimationNameByIndex can return the name of an animation to the animation controller by using the passed array index to call CAnimationItem::GetName.

```
LPCTSTR CAnimationItem::GetName( ) const
{
    return m_strName;
}
```

CAnimationItem::GetSRT

This method is, in many ways, the backbone of the entire D3DX animation system. It is responsible for generating the SRT animation data for a specified periodic position.

The animation controller will call the GetSRT method of an animation set for each animation in that set. Each time, the animation index will be passed and the function will generate the SRT data for that animation. This information will be returned to the controller and used to rebuild the frame matrix which shares the same name as the animation.

Although the animation controller expects the call to the GetSRT method of the animation set interface to generate the SRT data, we saw earlier that the animation set GetSRT method acts as a wrapper that forwards the request to the correct CAnimationItem:

```
    return m_pAnimations[ AnimationIndex ].GetSRT( PeriodicPosition, pScale,
                                                   pRotation, pTranslation );
```

As the above code snippet demonstrates, the CAnimationSet::GetSRT method uses the controller supplied index (identifying which animation it needs SRT data for) and periodic position to call the GetSRT method of the corresponding CAnimationItem. CAnimationItem::GetSRT performs the keyframe interpolation for the specified time and stores the final scale, rotation, and translation information in the output parameters.

We took a look at how to implement a GetSRT function in the textbook, so the code to this function should look very familiar to you. Since the function is rather large, we will discuss it a section at a time.

The first section is shown below followed by a discussion of the code.

```
HRESULT CAnimationItem::GetSRT( DOUBLE PeriodicPosition, D3DXVECTOR3 *pScale,
                                D3DXQUATERNION *pRotation,
                                D3DXVECTOR3 *pTranslation )
{
    ULONG                i;
    D3DXQUATERNION       q1, q2;
    double               fInterpVal, fTicks;
    LPD3DXKEY_VECTOR3    pKeyVec1 , pKeyVec2;
    LPD3DXKEY_QUATERNION pKeyQuat1, pKeyQuat2;

    // Validate parameters
    if ( !pScale || !pRotation || !pTranslation ) return D3DERR_INVALIDCALL;

    // Clear them out as D3D does for us :)
    *pScale       = D3DXVECTOR3( 0.0f, 0.0f, 0.0f );
    *pTranslation = D3DXVECTOR3( 0.0f, 0.0f, 0.0f );
    D3DXQuaternionIdentity( pRotation );

    // Reset index caching if we've wrapped around
    if ( PeriodicPosition < m_fLastPosRequest )
    {
        m_nLastScaleIndex       = 0;
        m_nLastRotateIndex      = 0;
        m_nLastTranslateIndex   = 0;

    } // End if wrapped / move backwards

    // Store the last requested position
    m_fLastPosRequest = PeriodicPosition;

    // Now calculate the 'TimeStamp' value.
    fTicks = PeriodicPosition * m_fTicksPerSecond;
```

The function is called by CAnimationSet::GetSRT and is passed the periodic position of the parent animation set along with the addresses of two 3D vectors and a quaternion which the function will populate with the interpolated SRT data. If any of these pointers are NULL, then the function returns immediately, reporting an invalid call.

Next, we set the passed scale and translation vectors to zero and set the rotation quaternions to identity. This is done for safety -- the animation might not have S, R, or T arrays defined, so default values will need to be returned.

The next portion of the above code then tests to see if the current periodic position is smaller than the periodic position that was stored in the previous call. If so, then the periodic position has wrapped around (looped or ping-ponged). It is very important for us to trap this occurance because it affects our bounding keyframe search optimization. Recall that we store the first of the two bounding keyframes (from each array) so that the next time the function is called, we do not have to start looping through the keyframes from the first element. For example, if the first time this function was called, we found the two bounding scale keyframes to be at positions 20 and 21 in the scale keyframes array, the value of 20 (the low bounding keyframe) would be stored in the m_nLastScaleIndex member variable. This means that the next time GetSRT is called for this animation, we can begin searching for bounding keyframes from position 20 in the array instead of 0, thereby skipping tests for the first 20 keyframes in the scale keyframe array.

As you can see, we store a 'last index' for each of the arrays (scale, rotation and translation). This optimization assumes a forward moving timeline, so it expects the next call to GetSRT to have a greater periodic position than the previous one. However, when the periodic position of the animation set loops around to zero again, this will not be the case.

Imagine that the first time GetSRT is called the periodic position is such that the low bounding scale keyframe used is at index 99 in our scale keyframe array. At the end of this function, we would store this index in the m_nLastScaleIndex member. Now imagine that the periodic position loops around the next time this function is called such that the low bounding keyframe should now be at index 0. If we simply started searching from index 20 to the end of the array we would never find any bounding scale keyframes. Therefore, you can see that we also cache the previous periodic position. If it is larger than the current periodic position passed into the function, it means that the periodic position has either looped around to zero or, in the case of ping-pong mode, the periodic position is moving backwards. When this is the case, we must start searching for bounding keyframes in each of our arrays from position zero again to make sure we test them all. We do this by setting the three previous keyframe index caches to zero.

The implication is that every time the periodic position loops around, we will not benefit from our 'last cached index' optimization. It also means that this optimization will provide no benefit for animation sets that are moving backwards. If the current periodic position has not looped around such that it is smaller than the previous position, no action is taken. You will see in a moment how the cached last keyframe indices will be used to efficiently search through the keyframe arrays.

Next we set the current periodic position in the m_fLastPosRequest member so that the next time this function is called, we will be able to detect a periodic position wrap-around (as just discussed).

The final section of the previous code takes the input periodic position and multiplies it by the TicksPerSecond ratio so that we have the periodic position in ticks rather than seconds. This step is important because the keyframe timestamps are defined in ticks and we want to find the two keyframes in each array that bound the periodic position in ticks, not seconds. Because we took the liberty of

storing the TicksPerSecond ratio of the parent animation set in a CAnimatonItem member variable, we can access it easily without having to implement a CAnimationSet accessor function to return this value every time we need it.

The next section of code is responsible for searching for the bounding keyframes in the animation's scale keyframe array. Notice that we do not simply loop from the start of the keyframe array every time, but start at the m_nLastScaleIndex element in the array. As discussed, this index will usually store the low bounding scale keyframe index from a previous call to the function. If the periodic position has looped around however, this value will be set to zero and the scale keyframe array is searched from the beginning. Notice that we also use two D3DXKEY_VECTOR3 pointers to point to the two bounding keyframes that we find.

```
    // ******************************************************************
    // * SCALE KEYS
    // ******************************************************************
    pKeyVec1 = pKeyVec2 = NULL;
    for ( i = m_nLastScaleIndex; i < m_nScaleKeyCount - 1; ++i )
    {
        LPD3DXKEY_VECTOR3 pKey     = &m_pScaleKeys[i];
        LPD3DXKEY_VECTOR3 pNextKey = &m_pScaleKeys[i + 1];

        // Do these keys  bound the requested time ?
        if ( fTicks >= pKey->Time && fTicks <= pNextKey->Time )
        {
            // Update last index
            m_nLastScaleIndex = i;

            // Store the two bounding keys
            pKeyVec1 = pKey;
            pKeyVec2 = pNextKey;
            break;

        } // End if found keys

    } // Next Scale Key
```

In the above code we loop and fetch the current scale keyframe and the one that follows it in the array. We then compare the periodic position in ticks (fTicks) to see if it is greater than or equal to the current keyframe's timestamp and less than or equal to the next keyframe's timestamp. If this condition is true, then we have found the two keyframes in the array that bound the current periodic position. When this is the case, we store the index of the current keyframe 'i' in the m_nLastScaleIndex member so that the next time this function is called, we can begin searching from this index. We then assign the local variables pKeyVec1 and pKeyVec2 to point to these two scale keyframes. Our search is complete and we have found the two bounding scale keyframes, so we exit the loop. The next step will be to interpolate between these two keyframes to generate a single scale vector that can be returned back to the controller and contribute to the building of the associated frame matrix.

As discussed in the textbook, we can use the current periodic position (in ticks) to perform a linear interpolation (or 'lerp' for short) between the two vectors. First we subtract the periodic position from the timestamp of the low bounding keyframe. Then we divide this value by the difference between the

two timestamps of the keyframes. This will result in a value between 0.0 and 1.0 which describes, as a percentage, the periodic position between these two keyframes. If a value of 0.25 was generated, it would mean that actual periodic position is ¼ of the way between timestamp one and timestamp two.

We feed this value into the D3DXVec3Lerp function which will return a vector that is a weighted interpolation of the two vectors. Using our 0.25 example, D3DXVec3Lerp would return a scale vector that is comprised of ¾ of keyframe one and ¼ of keyframe two. The basic formula for a lerp is ***Vector1 + t(Vector2 – Vector2)*** where **t** is our interpolation value between zero and one. The resulting interpolated scale vector is stored in the pScale vector that was passed in as a parameter by the animation controller. Below we see the code that performs this task:

```
    // Make sure we found keys
    if ( pKeyVec1 && pKeyVec2 )
    {
        // Calculate interpolation
        fInterpVal  = fTicks - pKeyVec1->Time;
        fInterpVal /= (pKeyVec2->Time - pKeyVec1->Time);

        // Interpolate!
        D3DXVec3Lerp( pScale,
                      &pKeyVec1->Value,
                      &pKeyVec2->Value,
                      (float)fInterpVal );
    } // End if keys were found
    else
    {
        // Scale is the same as the last scale key found
        if ( m_nScaleKeyCount )
          *pScale = m_pScaleKeys[ m_nScaleKeyCount - 1 ].Value;

        // Inform cache that it should no longer
        // search unless cache is invalidated
        m_nLastScaleIndex = m_nScaleKeyCount - 1;


    } // End if no keys found
```

If pKeyVec1 or pKeyVec2 are NULL then the two bounding keyframes could not be found in the scale array. For example, this might happen if the periodic position is outside the range of defined scale keys. When this is the case the ‘else’ block of the above code is executed. In this code block we first test if any scale keys even exist in this object and if so, then we assume that the periodic position is larger than the last defined scale keyframe and we just copy the last scale vector defined in the SRT array into pScale. Essentially this amounts to freezing the scaling animation at the last keyframe. Any periodic positions past this timestamp will generate the same final scale vector. We then set the last scale keyframe index to the final keyframe in the scale array so that the next time the function is called, we will not waste time looping and testing any scale keys -- we will just return the final scale key. We are basically stating that this animation appears to be over so we will just return the last scale keyframe unless the periodic position wraps around. If the periodic position does loop around, then we know that the last index caches are cleared and the entire keyframe array is used for bounding keyframe searches again.

In the next section of the code, we perform nearly an identical search and interpolation to generate the rotation quaternion. We search the rotation keyframe array for the two bounding keyframes and once found, perform a spherical linear interpolation (or 'slerp' for short) using the periodic position. This generates a final quaternion that is stored in the pRotation output parameter passed into the function by the animation controller. Here is the code that searches the quaternion array for the two bounding keyframes:

```
    // ******************************************************************
    // * ROTATION KEYS
    // ******************************************************************
    pKeyQuat1 = pKeyQuat2 = NULL;
    for ( i = m_nLastRotateIndex; i < m_nRotateKeyCount- 1; ++i )
    {
        LPD3DXKEY_QUATERNION pKey     = &m_pRotateKeys[i];
        LPD3DXKEY_QUATERNION pNextKey = &m_pRotateKeys[i + 1];

        // Do these keys bound the requested time ?
        if ( fTicks >= pKey->Time && fTicks <= pNextKey->Time )
        {
            // Update last index
            m_nLastRotateIndex = i;

            // Store the two bounding keys
            pKeyQuat1 = pKey;
            pKeyQuat2 = pNextKey;
            break;

        } // End if found keys

    } // Next Rotation Key
```

Now that we have our two keyframes we perform a slerp between them. But first we flip the axes of our quaternion keyframes so that they are left-handed to match our coordinate system. The D3DXQuaternionConjugate function is used for that purpose. It takes the quaternion (x,y,z,w) and returns the quaternion conjugate (-x,-y,-z, w). We can now generate our fInterpVal value by mapping the periodic position with respect to the two keyframe timestamps into the 0.0 to 1.0 range and feed this value into the slerp function to generate the final rotation quaternion that the animation controller needs.

```
    // Make sure we found keys
    if ( pKeyQuat1 && pKeyQuat2 )
    {
        // Reverse 'winding' of these values
        D3DXQuaternionConjugate( &q1, &pKeyQuat1->Value );
        D3DXQuaternionConjugate( &q2, &pKeyQuat2->Value );

        // Calculate interpolation
        fInterpVal  = fTicks - pKeyQuat1->Time;
        fInterpVal /= (pKeyQuat2->Time - pKeyQuat1->Time);

        // Interpolate!
        D3DXQuaternionSlerp( pRotation, &q1, &q2, (float)fInterpVal );

    } // End if keys were found
```

```
    else
    {
        // Rotation is the same as the last key found
        if ( m_nRotateKeyCount )
        D3DXQuaternionConjugate(pRotation,
                                &m_pRotateKeys[ m_nRotateKeyCount - 1 ].Value );

        // Inform cache it should no longer search unless cache is invalidated
        m_nLastRotateIndex = m_nRotateKeyCount - 1;

    } // End if no keys found
```

As was the case with the scale array, we also have an 'else' code block that is executed if no bounding rotation keyframes are found. When this is the case the same optimization logic is applied. We simply return the conjugate of the last rotation keyframe in the array and set the m_nLastRotateIndex to point at the last element in the array. Subsequent calls to the function will no longer need to search for bounding quaternion keyframes and can simply return the final quaternion in the array. Only when the periodic position is reversed or looped will the last index cache be invalidated and searches through the quaternion array re-enabled.

The final section of the code searches for the two bounding translation keyframes and returns the interpolated position vector. This code is almost identical to the code that finds and interpolates between scale keyframes (they are both D3DXKEY_VECTOR3 arrays) so we should now fully understand the following code without discussion.

```
    // ******************************************************************
    // * TRANSLATION KEYS
    // ******************************************************************
    pKeyVec1 = pKeyVec2 = NULL;
    for ( i = m_nLastTranslateIndex; i < m_nTranslateKeyCount - 1; ++i )
    {
        LPD3DXKEY_VECTOR3 pKey     = &m_pTranslateKeys[i];
        LPD3DXKEY_VECTOR3 pNextKey = &m_pTranslateKeys[i + 1];

        // Do these keys bound the requested time ?
        if ( fTicks >= pKey->Time && fTicks <= pNextKey->Time )
        {
            // Update last index
            m_nLastTranslateIndex = i;

            // Store the two bounding keys
            pKeyVec1 = pKey;
            pKeyVec2 = pNextKey;
            break;

        } // End if found keys

    } // Next Translation Key

    // Make sure we found keys
    if ( pKeyVec1 && pKeyVec2 )
    {
        // Calculate interpolation
```

```
        fInterpVal  = fTicks - pKeyVec1->Time;
        fInterpVal /= (pKeyVec2->Time - pKeyVec1->Time);

        // Interpolate!
        D3DXVec3Lerp( pTranslation,
                      &pKeyVec1->Value,
                      &pKeyVec2->Value,
                      (float)fInterpVal );

    } // End if keys were found
    else
    {
        // Rotation is the same as the last key found
        if ( m_nTranslateKeyCount )
             *pTranslation = m_pTranslateKeys[ m_nTranslateKeyCount - 1 ].Value;

        // Inform cache it should no longer search unless cache is invalidated
        m_nLastTranslateIndex = m_nTranslateKeyCount - 1;

    } // End if no keys found

    // We're done
    return D3D_OK;
}
```

We have now implemented all the components of our custom animation set. We examined how to build the CAnimationSet class which will be registered with the animation controller and we also discussed the CAnimationItem class that will be used by CAnimationSet to store, manage and interpolate keyframe data. CAnimationSet will be used in many of our demos from this point onwards, so be sure that you understand how it all works. Although the implementation of our ID3DXAnimationSet derived object may not have been something you thought we would have to do, the ID3DXKeyframedAnimationSet bug left us little choice. However, this has proven to be a beneficial aside as it has allowed us to get a much deeper insight into how the D3DX animation system works under the hood. It also provides you with some foundation material should you choose to implement other forms of custom animations that use application generated data.

Now that we have our drop-in replacement for the ID3DXKeyframedAnimationSet, let us return to our discussions concerning adding animation support to our CActor class. These features will be used in Lab Project 10.1 to play back an animated X file.

Source Code Walkthrough - CActor

Adding animation support to CActor will be a trivial task for the most part. While many functions will be added to the class in this lab project, virtually all of them will be simple wrappers around calls to the animation controller. Since these methods wrap the animation controller methods discussed in detail in the textbook, they will not require extensive coverage. Most are functions that set or retrieve properties from the actor's animation controller. This allows an application using our CActor class to access all the methods of its underlying animation controller via the CActor interface.

Below we see an excerpt from CActor.h which shows the new member variables that have been added in this lab project. For the sake of readability, we have not listed method declarations because we will cover the methods later. For now, just observe the new member variables which have been highlighted in bold.

```
class CActor
{
public:
        //---------------- Member Functions Go Here( Not Shown ) ---------------
private:
    LPD3DXFRAME                 m_pFrameRoot;               // Root 'frame'
    LPD3DXANIMATIONCONTROLLER   m_pAnimController;          // controller
    LPDIRECT3DDEVICE9           m_pD3DDevice;               // Direct3D Device .
    CALLBACK_FUNC               m_CallBack[CALLBACK_COUNT]; // callbacks
    TCHAR                       m_strActorName[MAX_PATH];   // Actor Filename

    D3DXMATRIX                  m_mtxWorld;      // Currently active world matrix.
    ULONG                       m_nOptions;      // The requested mesh options.

    // Limits of the Animation Mixer
    ULONG                       m_nMaxTracks;
    ULONG                       m_nMaxAnimSets;
    ULONG                       m_nMaxAnimOutputs;
    ULONG                       m_nMaxEvents;
    ULONG                       m_nMaxCallbackKeys;
};
```

Not surprisingly, we have now added an ID3DXAnimationController pointer to our CActor. It will be assigned to the animation controller returned from the D3DXLoadMeshHierachyFromX function (see LoadActorFromX).

Four of the five new members at the bottom of the listing contain the limits of the animation controller currently being used by the actor. Note that m_nMaxCallbackKeys is not a limit of the animation controller; it is used by the application to tell the actor the maximum number of callback keys that it will register with a single animation set. Earlier, we talked about how this value will be used when the actor is loading an X file. It indicates the size of the temporary array of callback keys that will be passed to the callback key callback function. This is the array that the application will fill with callback key data and return to the actor. We will see this all happening in a moment.

We know from earlier discussion that the animation controller limits must be defined at creation time. Such limits define the maximum number of tracks that can be used on the animation mixer, the maximum number of animation sets that can be simultaneously registered with the controller, the number of matrices/frames that can be manipulated by the controller, and the maximum number of sequencer events that can be registered on the global timeline. In the textbook we also discussed that when we use D3DXLoadMeshHierarchyFromX to load an animated X file, D3DX will create the animation controller for us. This means that setting these controller properties at creation time is out of our hands in that case. The maximum limits will all be specified automatically when the X file is loaded.

When D3DXLoadMeshHierarchyFromX is used to load an X file, an animation controller will be returned and stored by the actor if animation data was contained in the file. The maximum number of

tracks on the controller will always be set to two. Having only two tracks to use for blending (i.e., only two animation sets can be simultaneously played and blended at any one time) might prove to be too limiting in certain cases. You will see in a moment that the CActor will expose a method to allow us to ratchet up these limits either before or after the X file has been loaded and the animation controller has been created. Behind the scenes, the CActor will clone the old animation controller (if one previously exists) to create a new one with the desired maximum limits, while maintaining the integrity of the original controller's animation data. Ultimately this means that the application is not restricted to the default limits of the animation controller created by D3DXLoadMeshHierarchyFromX.

While there are many new functions which are simple one-line wrappers around animation controller interface calls, a number of significant changes have taken place in functions that previously existed (e.g., LoadActorFromX and some of its new helper functions). These methods have been upgraded from the previous lab project to create, store, and potentially clone the animation controller if the limits of the controller are not as desired. Thus, we will cover the code changes to these methods as well.

An intuitive place to start examining the changes to CActor is in its constructor.

CActor::CActor()

By default, we set the animation controller pointer to NULL as this will not receive a meaningful assignment until the X file is loaded. This will happen when the application calls the CActor::LoadActorFromX method.

We also set the default maximum limits for the animation controller. For consistency, we set them to the same defaults used by the D3DXLoadMeshHierarchyFromX function. You will see in a moment how these variables can be changed by the application.

By default, we set the maximum number of animation sets the controller will support to 1, although this will be assigned a proper value as soon as the X file is loaded.

```
CActor::CActor()
{
    // Reset / Clear all required values
    m_pFrameRoot      = NULL;
    m_pAnimController = NULL;
    m_pD3DDevice      = NULL;
    m_nOptions        = 0;

    // Setup the defaults for the maximum controller properties
    // (Defaults to the same defaults D3DX provides).
    m_nMaxAnimSets      = 1;
    m_nMaxAnimOutputs   = 1;
    m_nMaxTracks        = 2;
    m_nMaxEvents        = 30;
    m_nMaxCallbackKeys  = 1024;

    ZeroMemory( m_strActorName, MAX_PATH * sizeof(TCHAR) );
    D3DXMatrixIdentity( &m_mtxWorld );

    // Clear structures
```

```
    for ( ULONG i = 0; i < CALLBACK_COUNT; ++i )
        ZeroMemory( &m_CallBack[i], sizeof(CALLBACK_FUNC) );
}
```

Notice that we also set the maximum number of callback keys that can be registered with an animation set to 1024. This will probably be enough for most cases, but if this is not the case, we will be able to change this setting using the SetActorLimits method which we will discuss shortly. Just as we did in the previous lab project, we also set the world matrix of the actor to an identity matrix and zero out the name of the actor.

Finally, we loop through the actor's callback function array and set all elements to zero. This array previously contained only the callback functions for texture and materials, but we now have four elements in this array, with the fourth element reserved for registration of the callback key collection callback function. The actor defines the following enumeration for callback functions. It tells us the number of possible callbacks as well as the array indices for the callback function pointers.

```
enum CALLBACK_TYPE { CALLBACK_TEXTURE = 0,
                     CALLBACK_EFFECT = 1,
                     CALLBACK_ATTRIBUTEID = 2,
                     CALLBACK_CALLBACKKEYS = 3,
                     CALLBACK_COUNT = 4 };
```

The fourth array element stores a function pointer (and context data pointer) to an application defined callback function that will fill the callback key array for a given animation set. This callback function was discussed earlier in this workbook. As before, these callbacks are registered using the actor's RegisterCallback function which is unchanged from the previous lab project.

CActor::Release()

As in our previous lab project, the actor's Release method is called by the destructor to release all memory allocated by the actor. It may also be called by the application or from other actor methods when the data currently managed by the actor needs to be cleaned out or reset. It releases the frame hierarchy, the animation controller, and the device interface that it has stored. The two new function calls that have been added to this function are shown below in bold.

```
void CActor::Release()
{
    CAllocateHierarchy Allocator( this );

    // Release any active callback data
    ReleaseCallbackData( );

    // Release objects (notice the specific method for releasing the root frame)
    if ( m_pFrameRoot      ) D3DXFrameDestroy( m_pFrameRoot, &Allocator );
    if ( m_pAnimController ) m_pAnimController->Release();
    if ( m_pD3DDevice      ) m_pD3DDevice->Release();

    // Reset / Clear all required values
    m_pFrameRoot      = NULL;
```

```
    m_pAnimController = NULL;
    m_pD3DDevice      = NULL;
    m_nOptions        = 0;

    ZeroMemory( m_strActorName, MAX_PATH * sizeof(TCHAR) );

    // Since 'Release' is called just prior to loading, we should not
    // clear ANYTHING which is designed to be setup and stored prior
    // to the user making that load call. This includes things like
    // the world matrix, and the callbacks.
}
```

Releasing the animation controller interface is of course a logical thing to do when the actor is destroyed. Its release will, in turn, destroy all of its animation sets. However, before this is done, we must make sure that the callback data we have assigned to our animation sets is released. Remember, each callback key in our lab project will have a CActionData object assigned to its context data pointer. When the animation controller releases the animation sets it manages, it will simply release the callback array contained inside that animation set. Since this is the only place the CActionData pointers are stored, we will lose any ability after that point to de-allocate the memory containing the CActionData objects that were allocated in the callback key callback function at load time. Therefore, before we release the controller, the Release method makes a call to a new member function called ReleaseCallbackData. This function (covered next) will release the memory for each CActionData object referenced by callback keys in the callback array of each registered animation set.

It should be noted that while the CActor destructor simply wraps a call to the Release method, this method is also called from other places. For example, it is called by the actor prior to loading an X file to make sure that all existing data is released prior to the new data being loaded.

CActor::ReleaseCallbackData

There are two versions of this function in CActor (i.e., it is overloaded). This version of the ReleaseCallbackData method takes no parameters and is called from the CActor::Release method discussed above. It loops through each animation set currently registered with the controller and fetches an interface to that animation set using the ID3DXAnimationController::GetAnimationSet method. Once we have a pointer to the animation set interface, we call the overloaded version of this function ReleaseCallbackData, passing in the animation set interface as a parameter. This overloaded version of the function (which we will discuss in a moment) does the work of releasing the callback key context data contained in the animation set. The code to the version of the function that is called by CActor::Release is shown below.

```
void CActor::ReleaseCallbackData( )
{
    HRESULT             hRet;
    ULONG               i, SetCount;
    ID3DXAnimationSet * pSet = NULL;

    if ( !m_pAnimController ) return;

    // For each animation set, release the callback data.
```

```
    for (i=0,SetCount=m_pAnimController->GetNumAnimationSets(); i<SetCount; ++i )
    {
        // Retrieve the animation set
        hRet = m_pAnimController->GetAnimationSet( i, &pSet );

        if ( FAILED(hRet) ) continue;

        // Release them for this set
        ReleaseCallbackData( pSet );

        // release the anim set interface
        pSet->Release();

    } // Next Animation Set
}
```

After we have passed the animation set interface into the overloaded version of the function, we can assume that any callback key context data associated with the current animation set has been released. As such, the animation set interface can now safely be released. We do this for all animation sets registered with the animation controller so that on function return all callback context data associated with the actor has been cleaned up.

Keep in mind that this function does not release any animation sets from memory. The animation sets are unregistered later in the CActor::Release method when the animation controller is destroyed. All we are doing here is giving the actor a chance to remove CActionData objects associated with callback keys from memory.

CActor::ReleaseCallbackData (Overloaded Method)

This version of the method is called by the function previously discussed. It is passed the interface of an ID3DXAnimationSet derived class which, in our lab project, is a CAnimationSet interface since we replaced all ID3DXKeyframedAnimationSets with our own implementation. This function will release callback data using only the methods of the ID3DXAnimationSet interface, so it will work correctly with any animation set derived from that interface.

This function is tasked with looping through each callback key registered with the passed animation set and releasing its context data (if it exists). The actor is not concerned with what the context data pointer of a callback key is pointing at, only that it is pointing at some object derived from IUnknown. Since the actor makes this assumption, it can simply loop through each callback key in the array and, if the context data pointer of the current key is non-NULL, cast the void pointer to an IUnknown interface and call its Release method.

We discussed the implementation of our CActionData class earlier and saw how its release method decremented the internal reference count and deleted the object (itself) from memory when the count hit zero. We also examined how the CActionData object is created in the callback key callback function and that when the CActionData object is returned to the actor by the callback function, the actor does not increment its reference count. Our application does not store any copies of this CActionData interface or increment its reference count anywhere else in the application. Each CActionData object will have a

reference count of 1 when first registered with the actor. Thus, when the actor calls the release method for each of its CActionData objects, the objects will delete themselves from memory.

This method works properly with any context data that is an IUnknown derived class. Therefore, you do not have to use our CActionData class; you can use your own context data objects and custom classes if you wish. Just make sure that they are derived from IUnknown and that you correctly implement the methods of the IUnknown interface.

Here is the code to the function that releases all context data for the callback keys registered with a single animation set.

```
void CActor::ReleaseCallbackData( ID3DXAnimationSet * pSet )
{
    double     Position  = 0.0;
    ULONG      Flags     = 0;
    IUnknown * pCallback = NULL;

    // Keep searching until we run out of callbacks
    for ( ;SUCCEEDED( pSet->GetCallback( Position,
                                         Flags,
                                         &Position,
                                         (void**)&pCallback )) ; )
    {
        // Ensure we only get the first available (GetCallback will loop forever).
        if ( Position >= pSet->GetPeriod() ) break;

        // Release the callback (it is required that this be an
        // instance of an IUnknown derived class)
        if ( pCallback ) pCallback->Release();

        // Exclude this position from now on
        Flags = D3DXCALLBACK_SEARCH_EXCLUDING_INITIAL_POSITION;

    } // Next Callback
}
```

The function loops through each callback key in the animation sets callback key array using the ID3DXAnimationSet::GetCallback method. The first time this method is called we include the initial position in the search (position 0.0). The function returns the track position and callback key context pointer for the next scheduled callback key.

When we covered the GetCallback method earlier, we learned that it will continue to loop when searching for keys. For example, if the period of the animation set was 10 seconds and the last callback key returned was the one at 10 seconds, the next time the function is called it would wrap around to the beginning of the array and return the first callback key again. We do not want this behavior in this case because we only need a single pass through the callback key array to release the context data. Thus we break from the loop as soon as the position variable is greater than the period of the animation. You can see that this position variable starts at zero, and is passed into each call to GetCallback method so that its value is overwritten by the position of the next callback key that is scheduled. As soon as this position reaches the length of the animation set (remember that the function returns a position analogous to a

non-looping track time), we know that we have processed all the keys and released any context data that needs releasing.

If we have not yet reached the period of the animation set, then we have a context data pointer returned for the next key in the list. If this is not NULL then it means that it must point to an object derived from IUnknown (our CActionData object for example). We simply cast this void pointer to an IUnknown interface pointer and call its release method. This will decrement the object reference count and cause it to delete itself from memory (unless you have incremented its reference count and stored pointers to these objects somewhere else in your application).

Finally, notice that after the first call to GetCallback is issued, for all future calls we pass in the `D3DXCALLBACK_SEARCH_EXCLUDING_INITIAL_POSITION` flag so that we do not get back the callback data we have just processed.

CActor::SetActorLimits

The application should have the ability to change the maximum limits of the actor's underlying controller both before and after the animation data has been loaded from an X file. This allows it to load an X file and then increase properties like the available mixer tracks from 2 (the default) to however many are desired.

Since this function can be called prior to or after the application has called the CActor::LoadActorFromX function, it works differently depending on whether or not the actor's animation controller already exists. If the controller already exists, then the new requested maximums are stored in the actor's member variables and are compared against the limits of the current controller. If any of the requested maximums are greater than what the current controller can handle, then this function will clone the animation controller into one that can support *at least* the requested maximums. The words 'at least' are important here because this function will only increase the limits of its internal animation controller; it will never decrease them.

If the application passes in zero for any of the properties, the maximum extents of the current controller for that property will be inherited from the previous controller by the clone. For example, if we pass in a MaxTracks parameter of zero but the controller already supports two, the cloned controller will support two tracks also. What is important is the fact that the while passed maximum extents are stored in the actor's member variables, the controller will only need to be cloned if one or more of the requested extents is greater than what is currently supported by the actor's animation controller.

The other mode of this function handles the case when the controller does not currently exist. This can happen if the application has called this function prior to calling LoadActorFromX. When this is the case, the maximum extents will simply be stored in the actor's member variables and the function will return. When the application later calls LoadActorFromX, the function will clone the animation controller automatically if the animation controller returned to the actor from the D3DXLoadMeshHierarchyFromX function does not support the previously set maximum extents the application desires.

The exception to the parameter list is the final parameter, MaxCallbackKeys. This value has nothing to do with the animation controller; it sets the size of the temporary callback key buffer that will be allocated in LoadActorFromX and passed to the application-defined callback key callback function. MaxCallbackKeys should be at least as large as the maximum number of callback keys your application intends to register with a single animation set. The default value is 1024, which was set in the constructor.

Let us now have a look at this function a section at a time.

```
HRESULT CActor::SetActorLimits(     ULONG MaxAnimSets /*= 0*/,
                                    ULONG MaxTracks /* = 0*/,
                                    ULONG MaxAnimOutputs /* = 0 */,
                                    ULONG MaxEvents /* = 0 */,
                                    ULONG MaxCallbackKeys /* = 0 */ )
{
    HRESULT hRet;
    bool    bCloneController = false;
    ULONG   Value;

    // Cache the current details
    if ( MaxAnimSets     > 0 ) m_nMaxAnimSets     = MaxAnimSets;
    if ( MaxTracks       > 0 ) m_nMaxTracks       = MaxTracks;
    if ( MaxAnimOutputs  > 0 ) m_nMaxAnimOutputs  = MaxAnimOutputs;
    if ( MaxEvents       > 0 ) m_nMaxEvents       = MaxEvents;
    if ( MaxCallbackKeys > 0 ) m_nMaxCallbackKeys = MaxCallbackKeys;

    // If we have no controller yet, we'll take this no further
    if ( !m_pAnimController ) return D3D_OK;
```

In this first section we test the passed parameters and if any are larger than zero, we take this as a request that the caller wants to change this property. Therefore, any parameter that is larger than zero is stored directly in the actor's member variables. If the controller is not yet created, then our work is done and we simply return. This will typically be all this function does when it is called from the application prior to the X file being loaded into the actor (or if the X file has no animation data, in which case, no controller will exist for the actor anyway).

If the controller is already created then we need to find out if we need to clone it. The first thing we do is copy the current maximum extents of the actor stored in its member variables into some local variables that we can resize later in the function. We reuse the parameter created variables for this purpose since the values passed in have either been stored in the actor (in which case we are just copying them straight back over again), or were zero (in which case we are copying over the previous limit the actor had set for that property). After this copy, we have the exact desired limits that this actor should be able to support in local variables that we can manipulate.

```
    // Store back in our temp variables for defaults
    MaxAnimSets     = m_nMaxAnimSets;
    MaxTracks       = m_nMaxTracks;
    MaxAnimOutputs  = m_nMaxAnimOutputs;
    MaxEvents       = m_nMaxEvents;
    MaxCallbackKeys = m_nMaxCallbackKeys; // Should never be zero
```

When we clone the animation controller, we must pass all the limits of the new controller into the cloning function; not just the ones we wish to change. This makes sure that all limits are inherited by the cloned controller. If the actor has a controller that was previously limited to two mixing tracks, we might call this method and simply pass in zero as this limit. This states that we are quite happy with the current limit and wish it to survive the clone. The local variables in the above code now contain the limits for each controller property that we would like in our cloned controller.

However, as mentioned earlier in this workbook, we only wish to clone the controller if the desired limits exceed those of the pre-existing controller. If the current controller already has limits that are equal to or exceed the requested limits, no clone takes place. The current controller will be able to handle the applications requirements in that case.

The next section of code extracts each limit from the current controller and tests to see if it is smaller than the requested limit. If it is, then the controller will need to be cloned, so we set the bCloneController Boolean to true. If the requested limit is less than or equal to the current limit of the controller, we simply set the local limit variable to the current limit of the controller.

```
    // Otherwise we must clone if any of the details are below our thresholds
    Value = m_pAnimController->GetMaxNumAnimationOutputs();

    if ( Value < MaxAnimOutputs )
        bCloneController = true; else MaxAnimOutputs = Value;

    Value = m_pAnimController->GetMaxNumAnimationSets();

    if ( Value < MaxAnimSets )
        bCloneController = true; else MaxAnimSets = Value;

    Value = m_pAnimController->GetMaxNumTracks();
    if ( Value < MaxTracks )
         bCloneController = true; else MaxTracks = Value;

    Value = m_pAnimController->GetMaxNumEvents();
    if ( Value < MaxEvents )
        bCloneController = true; else MaxEvents = Value;
```

To understand why this code is necessary, let us imagine that we have a controller that is currently capable of managing 100 animation outputs. Let us also imagine that the application calls this method and specifies an animation output limit of 80. In this case, the controller does not need to be cloned because we only wish to expand the limits of the controller, not contract them. We made this design decision to prevent the application from accidentally breaking the animation data loaded from the X file. In this example, the actor's requested limit (80) would be compared against the current limit of the controller (100) and the Boolean is not set to true. Instead we simply upgrade the requested limit (MaxAnimOutputs=80) to the current limit of the controller (MaxAnimOutputs=100).

While this might seem like a strange thing to do, remember that while this particular limit might not cause the controller to be cloned, one of the other limits of the controller may be lower than requested, causing the controller to be cloned anyway. By upgrading the local variable limits so that they are at least as large as the current controller, that limit will be inherited by the clone. If the MaxAnimSets

member contained a value of 10 but the current controller only supported 2 tracks, the controller would be cloned. However, when we pass the MaxAnimOutputs local variable into the clone function, we are specifying that the controller be created with 100 animation outputs (the original maximum) instead of 80 animation outputs (the requested amount). So this code is necessary because we want to make sure that the limits of the original controller that will not change survive in the cloned controller. If we did not have this safeguard, and we cloned the animation controller with 80 animation outputs instead of 100, the animation data for 20 frames in the hierarchy would be lost in the clone. This is almost always something that the application would not intend to happen.

At this point in the code, our Boolean variable will indicate that the controller either does or does not need cloning. If it does then we pass our local variables as the limits to the ID3DXAnimationController::CloneAnimationController method to create a new controller with the desired limits. All animation data that existed in the original controller will now be present in the clone (even our context data pointers).

Once the clone has been performed, we release the original controller's interface and assign the actor's m_pAnimController pointer to the newly cloned interface. The remaining section of the code that performs the clone is shown below.

```
    // Clone our animation controller if there are not enough set slots available
    if ( bCloneController )
    {
        LPD3DXANIMATIONCONTROLLER pNewController = NULL;

        // Clone the animation controller
        hRet = m_pAnimController->CloneAnimationController( MaxAnimOutputs,
                                                            MaxAnimSets,
                                                            MaxTracks,
                                                             MaxEvents,
                                                           &pNewController );
        if ( FAILED(hRet) ) return hRet;

        // Release our old controller
        m_pAnimController->Release();

        // Store the new controller
        m_pAnimController = pNewController;

    } // End if requires clone

    // Success
    return D3D_OK;
}
```

CActor::LoadActorFromX

The LoadActorFromX function is another method that existed in the previous lab project but has now been upgraded to facilitate the incorporation of animation data. The code that has been added is highlighted in bold. The changes may look trivial but this is because the method performs most of the new functionality in new member functions of the actor. Let us look at it a bit at a time.

```
HRESULT CActor::LoadActorFromX( LPCTSTR FileName, ULONG Options,
                                LPDIRECT3DDEVICE9 pD3DDevice,
                                  bool bApplyCustomSets /* = true */ )
{
    HRESULT hRet;
    CAllocateHierarchy Allocator( this );

    // Validate parameters
    if ( !FileName || !pD3DDevice ) return D3DERR_INVALIDCALL;

    // Release previous data!
    Release();

    // Store the D3D Device here
    m_pD3DDevice = pD3DDevice;
    m_pD3DDevice->AddRef();

    // Store options
    m_nOptions = Options;

    // Load the mesh heirarchy and the animation data etc.
    hRet = D3DXLoadMeshHierarchyFromX( FileName, Options, pD3DDevice, &Allocator,
                                       NULL, &m_pFrameRoot, &m_pAnimController );
    if ( FAILED(hRet) ) return hRet;

    // Copy the filename over
    _tcscpy( m_strActorName, FileName );
```

This first section of the code is almost entirely unchanged. It simply increases the reference count of the passed Direct3D device and stores its pointer in a member variable. It then uses the D3DXLoadMeshHierarchyFromX function to load the frame hierarchy. The only update here is that we now pass the address of an ID3DXAnimationController interface pointer to the loading function. If the X file contains animation data then an animation controller will be created for us along with its animation sets and it will be returned to us via this pointer. At this point, we will have a pointer to the animation controller interface for our hierarchy stored in the actor's m_pAnimationController member variable.

We then copy the name of the X file from which the actor was loaded into the m_strActorName member variable. We saw in the previous lab project that the application uses the actor's name when processing X file references inside an IWF file. If an X file has already been loaded, then an actor with the same name as the X file will already exist in the scene. The application can then choose to instance that actor instead. Now however, the actor name also plays another role. We discussed earlier that the application defined callback function for feeding in callback keys to the actor will also be passed the name of the

actor. This allows the application defined callback to test whether this is an X file it wishes to define callback keys for.

The next section of code is new. The first thing the actor does once it knows it has a valid controller is call the CActor::SetActorLimits method:

```
    // Apply our derived animation sets
    if ( m_pAnimController )
    {
        // Apply our default limits if they were set before we loaded
        SetActorLimits( );
```

This time, the function is passed no parameters, indicating that the parameters passed into the function will all be zero by default. When we discussed this function earlier we learned that when a given limit is zero, the current value for that limit stored in the actor's member variables is used. If the application has used the SetActorLimits method to set its desired limits for the controller prior to calling the LoadActorFromX function, the clone will be performed if the controller created for us by D3DX is not capable of fulfilling our application requirements.

The next line of code is very important. It tests to see if a callback key callback function has been registered with this actor by the application. If it has, then the pFunction pointer of the CALLBACK_FUNC structure in the fourth element (CALLBACK_CALLBACKKEYS = 4) of the actor's callback function array will not be NULL. If this is the case, a new method, which we have not yet discussed, is called. This method is called ApplyCallbacks and will be discussed next.

The ApplyCallbacks function will loop through every animation set currently registered with the controller, and for each one, will call the application defined callback function. The callback function will be passed a temporary array of D3DXKEY_CALLBACK structures. The application can then store callback keys and context data in this array before returning it back to the ApplyCallback method. This method will then replace the current animation set with a new copy of the animation set that has the callback keys registered with it. Remember, we have to do this because we have no way of registering callback keys with an animation set after it has been created.

```
        // If there is a callback key function registered, collect the callbacks
        if ( m_CallBack[CALLBACK_CALLBACKKEYS].pFunction != NULL )
             ApplyCallbacks();
```

We will look at the ApplyCallbacks method in a moment, but just note that when it returns, our animation controller may now contain animation sets that have callback keys registered with them.

The final section of the function code uses another new CActor method called ApplyCustomSets. As discussed earlier in this workbook, at the time of writing, a bug existed in the ID3DXKeyframedAnimationSet::GetSRT method. Because of this, we constructed a way around the problem by replacing all ID3DXKeyframedAnimationSets registered with our controller with CAnimationSet copies. That is what the ApplyCustomSets method does (we will cover the code to this method in a moment). It loops through every animation set currently registered with the controller and creates a replacement CAnimationSet object for each. It then copies over all the animation data from the

original keyframed animation set into our own CAnimationSet. Once we have a copy of each animation set, we release the original keyframed animation sets and register our CAnimationSet objects in their place. We covered all of the code for CAnimationSet earlier in this workbook.

```
        // Apply the custom animation set classes if we are requested to do so.
        if ( bApplyCustomSets == true )
        {
            hRet = ApplyCustomSets();
            if ( FAILED(hRet) ) return hRet;

        } // End if swap sets

    } // End if any animation data

    // Success!!
    return D3D_OK;
}
```

Notice that the LoadActorFromX method accepts a Boolean parameter called bApplyCustomSets. Only if this value is set to true will the ApplyCustomSets method be called and our own CAnimationSet implementations replace the ID3DXKeyframedAnimationSets registered with the controller. This parameter allows us to quickly unplug our CAnimationSet objects from the system when the bug in ID3DXKeyframedAnimationSet is fixed in the future.

While only a small amount of code was added to LoadActorFromX, there were calls made to two new methods which we will look at next. These two new methods do contain significant amounts of code.

CActor::ApplyCallbacks

The ApplyCallbacks method is called by LoadActorFromX after the actor has been loaded and already has an animation controller. The purpose of this function is to provide the application with a means of registering callback keys for animation sets registered with the animation controller. Keys cannot be registered with an animation set after it has been created, so we provide this functionality by making new copies of the animation sets for which the application has specified callback keys. We can register these new callback keys as we create the new animation set copies and then release the original animation sets and register the new ones (with the callback keys) in their place.

The first section of this function is shown next. It checks that the animation controller of the actor is valid and that it already has registered animation sets. If not, then the function simply returns. Next, we store (in the SetCount local variable) the number of animation sets currently registered with the animation controller. We will need to loop through each animation set and call the application defined callback function (if it has been registered) to give the application a chance to specify callback keys for each animation set.

Once we have recorded the number of sets in the animation controller, we allocate a temporary array of D3DXKEY_CALLBACK structures. For each animation set processed, we will pass this array into the application defined callback function. The callback function can then fill this array with callback data as we saw earlier in this workbook. Notice that the size of this temporary array is determined by the actor limit m_nMaxCallbackKeys, which can be changed using the SetActorLimits method. This is initially

set to 1024, which means the callback function will have enough room to register 1024 callback keys in this array for a single animation set. Remember, this is a per-set limit since the temporary buffer will be reused for each animation set processed by this function. The m_nMaxCallbackKeys member variable allows us to control the temporary memory overhead used by this function.

```
HRESULT CActor::ApplyCallbacks( )
{
    ULONG   i, j;
    HRESULT hRet;
    ID3DXKeyframedAnimationSet * pNewAnimSet = NULL, **ppSetList = NULL,
                               * pKeySet = NULL;
    ID3DXAnimationSet          * pAnimSet    = NULL;
    D3DXKEY_CALLBACK           * pCallbacks  = NULL;

    // Validate
    if ( !m_pAnimController || !m_pAnimController->GetNumAnimationSets( ) )
         return D3DERR_INVALIDCALL;

    // Store set count
    ULONG SetCount = m_pAnimController->GetNumAnimationSets( );

    // Allocate our temporary key buffer
    pCallbacks = new D3DXKEY_CALLBACK[ m_nMaxCallbackKeys ];
    if ( !pCallbacks ) return E_OUTOFMEMORY;

    // Allocate temporary storage for our new animation sets here
    ppSetList = new ID3DXKeyAnimationSet*[ SetCount ];

    if ( !ppSetList ) { delete pCallbacks; return E_OUTOFMEMORY; }
```

Once we have allocated the callback key array, we allocate an array of ID3DXKeyAnimationSet interface pointers. The size of this array is equal to the number of animation sets currently registered with the controller.

We have several things we must consider when implementing this function. First, we will need to unregister any animation sets that need to be replaced. These will only be animation sets that have a callback key count greater than zero. For any others we will let them remain registered with the controller. Second, we may have a situation where the controller has some animation sets that are not of the ID3DXKeyframedAnimationSet type (e.g., custom sets, compressed animation sets, etc.). Since we can only register callback keys with keyframed animation sets, we will want to also leave these others alone, registered with the controller. To ease this process, we have decided to implement this in three passes/loops. This makes the code slightly longer but easier to understand and maintain.

In the first loop, we will fetch each animation set from the animation controller and store a copy of its interface pointer in the temporary ID3DXAnimationSet array we have just allocated. In the loop, we will call ID3DXAnimationController::GetAnimationSet to fetch each set. As this function returns a pointer to the base interface, this is ideal. We simply store pointers to all the animation sets (keyframed or otherwise) in our temporary animation set array and then remove that set from the controller. What we will have at the end of this first pass is an empty animation controller, with pointers to each animation set it used to have registered in our temporary array. We can then work directly with this array, rejecting

or replacing animation sets we no longer want and leaving the ones we do want unaltered. At the end of the function, we can loop through this array and register all the animation sets back with the controller. Some of these animation sets will have been replaced by copies containing callback key data.

Here is the code that performs this first un-register pass.

```
for ( i = 0; i < SetCount; ++i )
    {
        // Keep retrieving index 0 (remember we unregister each time)
        hRet = m pAnimController->GetAnimationSet( 0, &pAnimSet );
        if ( FAILED( hRet ) ) continue;

        // Store this set in our list, and AddRef
        ppSetList[ i ] = pAnimSet;
        pAnimSet->AddRef();

        // Unregister the old animation sets first otherwise we won't have enough
        // 'slots' for the new ones.
        m_pAnimController->UnregisterAnimationSet( pAnimSet );

        // Release this set, leaving only the item in the set list.
        pAnimSet->Release();

    } // Next Set
```

Notice that when we copy the interface pointer for each animation set, we increase its reference count. If you study the previous code, you will realize that prior to the GetAnimationSet call, the animation controller is the only object that has an interface to each animation set, so the reference count of the animation set would be 1. When we call GetAnimationSet, the controller returns an interface to our application, automatically increasing the reference count before returning the pointer. The reference count of the animation set at this point is 2. We then copy this interface pointer into our array and increase the reference count again, so the reference count is 3. This is correct since there are three pointers to this interface currently in use. The animation controller has its own pointer to it, the pAnimSet local variable currently has a reference to it, and our interface pointer array has the third. We then call the animation controller's UnregisterAnimatonSet method so that the animation set releases its pointer to the interface, taking the reference count down to 2. Next we release the temporary pAnimSet pointer which decrements the reference count to 1. This is now exactly how we want it -- the only reference to this animation set is now in our local ID3DXAnimationSet array.

The next pass through the animation sets is where all the work is done. We loop through each animation set in our temporary array and query its interface. If it is not an ID3DXKeyframedAnimationSet interface, we will skip to the next iteration of the loop. We are essentially leaving this animation set alone and will re-add it to the controller later in the function. It will be unaltered from its initial state. If it is a keyframed animation set, then the QueryInterface method will return us a pointer to that interface:

```
    // Loop through each animation set and rebuild a new one containing the
    // callbacks.
    for ( i = 0; i < SetCount; ++i )
    {
```

```
        // Skip if we didn't manage to retrieve
        if ( ppSetList[i] == NULL ) continue;

        // Retrieve the set we're working on
        pAnimSet = ppSetList[i];

        // Query the interface to get a keyframed animation set
        hRet = pAnimSet->QueryInterface( IID_ID3DXKeyframedAnimationSet,
                                         (void**)&pKeySet );
        if ( FAILED(hRet) ) continue;
```

If we get this far in this iteration, we have a keyframed animation set which the application may wish to register callback keys for. We now have to call the application defined callback function to see if that is the case.

In the following code we see a technique we have used many times before when calling callback functions. If the application has registered a callback key callback function with the actor, its function pointer will exist in the fourth element in the actor's callback function array. If you examine CActor.h you will see that we have defined a type called COLLECTCALLBACKS which describes a pointer to this method:

```
typedef ULONG (*COLLECTCALLBACKS )( LPVOID pContext,
                                    LPCTSTR strActorFile,
                                    LPD3DXKEYFRAMEDANIMATIONSET pAnimSet,
                                    D3DXKEY_CALLBACK pKeys[] );
```

As you can see, any variables we declare with this type will be pointers to functions with the exact function signature of our application defined callback function (dictated by the actor). Your callback function *must* use this exact signature.

In the following code we create a pointer of this type called CollectCallbacks and assign it the value of the pFunction member of the fourth element in the actor's callback array. If the application has registered a callback function of this type with the actor (using the CActor::RegisterCallback method), then we will now have a pointer to the function that we can call. Otherwise, this element will be set to NULL, indicating the application has expressed no desire to register any callback keys with this actor's animation sets.

> **Note**: It is important to remember that the application must register any callback functions with the actor prior to calling LoadActorFromX because these functions are called in the middle of the loading process. This was also true of the texture and material callback functions that were covered in the previous lab projects.

In the next section of code, we fetch the function pointer and use it to invoke the callback function, passing any context data that was registered (the CScene object instance in our project). We also pass the function the name of the actor so that the callback function can determine which actor is making the call and, therefore, what callback data that actor should receive. We also pass in a pointer to the animation set interface so the callback function can extract information about which set it is being called for. In the example callback function we showed earlier, the animation set's interface was used to extract the name of the set. This allowed the callback function to determine what callback data should be

registered (if any) with the current animation set being processed. Finally, we pass in a pointer to our temporary D3DXKEY_CALLBACK array so that the application can fill it with callback data for the current animation set.

When the callback function returns, it will return an integer describing how many keys were placed in the array by the callback function. If the function returns zero, then we do not have to alter this animation set; we can simply continue to the next iteration of the loop and leave this animation set alone. This animation set will be re-registered with the animation controller later in the function. Here is the code that shows the process just described.

```
        // Retrieve our application's registered function pointer
        COLLECTCALLBACKS CollectCallbacks
            = (COLLECTCALLBACKS)m_CallBack[ CALLBACK_CALLBACKKEYS ].pFunction;

        // Request that our key buffer be filled
        ULONG KeyCount
            = CollectCallbacks( m_CallBack[ CALLBACK_CALLBACKKEYS ].pContext,
                                m_strActorName, pKeySet, pCallbacks );

        // If no keys are to be added, leave the existing one alone.
        if ( KeyCount == 0 ) { pKeySet->Release(); continue; }
```

If we get this far, then the current animation set we are processing is one that the application has supplied us with callback keys for. Therefore, as callback keys can only be registered with an animation set at creation time, we need to create a new copy of this animation set which will eventually be used to replace it. In the next section of code we create a new animation set that has the same name, source ticks per second, playback type, and number of animation as the animation set we are copying.

```
        // Create a new animation set for us to copy data into
        hRet = D3DXCreateKeyframedAnimationSet( pKeySet->GetName(),
                                                pKeySet->GetSourceTicksPerSecond(),
                                                pKeySet->GetPlaybackType(),
                                                pKeySet->GetNumAnimations(),
                                                KeyCount,
                                                pCallbacks,
                                                &pNewAnimSet );

        if ( FAILED(hRet) ) { pKeySet->Release(); continue; }
```

We now have a new animation set which has the same parameters as the original animation set, with one big difference. Notice that when we call the D3DXCreateKeyframedAnimationSet method, now we also pass in the number of callback keys (returned by the callback function) and a pointer to the array containing those keys (populated by the callback function). As result, the new animation set created will contain the callback key data supplied by the callback function.

Of course our job is not yet done. Although our new animation set shares the same properties as the original set and now includes callback data, it still contains no animation data. Although our new animation set has had space reserved for the correct number of animations, these animations have no data. We will have to manually extract the SRT keyframes from each animation in the original set and assign them to matching animations in the new set.

The following code performs the copying of the SRT data for each animation. It sets up a loop to iterate through each animation in the original set. For each animation, it first fetches the number of scale, rotate, and translate keys registered with that animation. These three values are then used to allocate temporary arrays to store copies of the SRT data for this animation. We then copy the scale, rotation and translation keys from the original animation into these three arrays.

```
        // Copy over the data from the old animation set
        //(we've already stored the keys)
        for ( j = 0; j < pKeySet->GetNumAnimations(); ++j )
        {
            LPCTSTR               strName = NULL;
            ULONG                 ScaleKeyCount, RotateKeyCount,
                                  TranslateKeyCount;
            D3DXKEY_VECTOR3     * pScaleKeys = NULL, * pTranslateKeys = NULL;
            D3DXKEY_QUATERNION  * pRotateKeys = NULL;

            // Get the old animation sets details
            ScaleKeyCount     = pKeySet->GetNumScaleKeys( j );
            RotateKeyCount    = pKeySet->GetNumRotationKeys( j );
            TranslateKeyCount = pKeySet->GetNumTranslationKeys( j );

            pKeySet->GetAnimationNameByIndex( j, &strName );

            // Allocate any memory required
            if ( ScaleKeyCount     )
               pScaleKeys     = new D3DXKEY_VECTOR3[ ScaleKeyCount ];

            if ( RotateKeyCount    )
              pRotateKeys    = new D3DXKEY_QUATERNION[ RotateKeyCount ];

            if ( TranslateKeyCount )
              pTranslateKeys = new D3DXKEY_VECTOR3[ TranslateKeyCount ];
            // Retrieve the keys
            if ( pScaleKeys     ) pKeySet->GetScaleKeys( j, pScaleKeys );
            if ( pRotateKeys    ) pKeySet->GetRotationKeys( j, pRotateKeys );
            if ( pTranslateKeys ) pKeySet->GetTranslationKeys( j, pTranslateKeys );
```

We now have the SRT data for the current animation we are copying over. Our next task is to register these SRT keys with the matching animation in the new animation set using the ID3DXKeyframedAnimationSet::RegisterAnimationSRTKeys method. When this function returns, D3DX will have copied the SRT data into the animation in the new set. These temporary SRT arrays are no longer needed and can be deleted.

```
            hRet = pNewAnimSet->RegisterAnimationSRTKeys(  strName,
                                                           ScaleKeyCount,
                                                           RotateKeyCount,
                                                           TranslateKeyCount,
                                                           pScaleKeys,
                                                           pRotateKeys,
                                                           pTranslateKeys, NULL );
            // Delete the temporary key buffers
            if ( pScaleKeys     ) delete []pScaleKeys;
            if ( pRotateKeys    ) delete []pRotateKeys;
```

```
            if ( pTranslateKeys ) delete []pTranslateKeys;

            // If we failed, break out
            if ( FAILED(hRet) ) break;

        } // Next Animation 'item'
```

If for some reason we got this far but were forced to break out of the above loop before each animation had been copied, then it means something went wrong during the copy procedure. The next section of code shows that if this is the case, we will release the new animation set and leave the original animation as is. We just continue to the next iteration of the loop to process the next animation set. This is a safe way to try to continue the process in the event of some unforeseen error.

```
        // If we didn't reach the end, this is a failure
        if ( j != pKeySet->GetNumAnimations() )
        {
            // Release new set, and reference to old
            pKeySet->Release();
            pNewAnimSet->Release();
            continue;

        } // End if failed
```

At this point we have copied over all the animation data from the original animation set into the new animation. The original animation set can now be released from memory as it will no longer be needed. Notice how we perform two releases to free the original animation set. The pKeySet pointer is the ID3DXKeyframedAnimationSet interface to the original animation set that we retrieved with the QueryInterface call. We then release the original animation set interface pointer stored in thre array and replace it with our new copy.

```
        // Release our duplicated set pointer
        pKeySet->Release();

        // Release the actual animation set, memory will now be freed
        pSetList[I]->Release();

        // Store the new animation set in our temporary array
        ppSetList[i] = pNewAnimSet;

    } // Next Set
```

Now every animation set has been processed. Some may have been replaced with copies, while others may have been left in their original state. Either way, at this point, the ppSetList array contains all the animation sets we wish to register with the animation controller. In the third pass we do exactly that. We loop through each animation set in this array and register it with the animation controller. We then release the interface pointer stored in the array since we no longer need it. This will not cause the animation set to be freed from memory because the animation controller has a pointer to it now and will have incremented its reference count when the set was registered.

```
    // third pass, register all new animation sets
```

```
    for ( i = 0; i < SetCount; ++i )
    {
        // Nothing stored here?
        if ( !ppSetList[i] ) continue;

        // Register our brand new set. (AddRef is called)
        m_pAnimController->RegisterAnimationSet( ppSetList[i] );

        // Release our hold on the custom animation set
        ppSetList[i]->Release();

    } // Next Set
```

The last section of the function frees the temporary animation set array and callback key array from memory. It also assigns the controller's first animation set to track zero by default. If we did not do this then no animation set would be assigned to any track. Remember, although the D3DXLoadMeshHierarchyFromX function automatically assigns an animation set to track zero on our behalf, we have unregistered all of those animation sets and undone this mapping.

```
    // Free up any temporary memory
    delete []ppSetList;
    delete []pCallbacks;

    // Set the first animation set into the first track, otherwise
    // nothing will play, since all previous tracks have potentially been
    //  unregistered.
    SetTrackAnimationSetByIndex( 0, 0 );

    // We're all done
    return D3D_OK;
}
```

That was a sizable function but ultimately very straightforward. We are essentially just cloning animation sets manually so that we can add additional data to them.

CActor::ApplyCustomSets

One of the last methods to be called from the LoadActorFromX method is CActor::ApplyCustomSets. This function gives us a chance to replace all ID3DXKeyframedAnimationSets with our own CAnimationSet objects. You will recall from our earlier discussion that this allows us to sidestep a bug in the ID3DXKeyframedAnimationSet::GetSRT method that existed at the time of this writing.

This method follows, for the most part, the same logic as the CActor::ApplyCallbacks method just discussed. In fact, it executes using the same three pass apporach. First it copies all the animation sets from the controller into a locally allocated array. Each animation set is then un-registered with the controller. In the second pass we loop through each animation set in the array. For each ID3DXKeyframedAnimationSet we find, we copy its information into a new CAnimationSet object. Luckily, the CAnimationSet constructor takes care of the actual copying of the keyframe data from the passed ID3DXKeyframedAnimationSet, so this code is a lot shorter than the previous function. For each

set we clone, we release the original. As before, if we find any compressed or custom animation sets we leave them alone in the array for re-registration later in the function. At the end of the second pass, we will have an array of animation sets where each keyframed animation set has been replaced by a CAnimationSet object. In the third pass, we simply register every animation set in this array with the animation controller.

Note however that this method is called by CActor::LoadActorFromX after the ApplyCallbacks method has been called. Therefore, the animation sets currently registered with the controller may well have callback keys defined for them. As we saw when we covered the CAnimationSet constructor, the callback keys of the animation set are also copied over into the new CAnimationSet copy.

The first pass of this function should look familiar. It allocates an array of ID3DXAnimationSet interfaces large enough to store a copy of each animation set currently registered with the controller.

```
HRESULT CActor::ApplyCustomSets( )
{
    ULONG   i;
    HRESULT hRet;
    CAnimationSet       * pNewAnimSet = NULL;
    ID3DXAnimationSet   * pAnimSet    = NULL, ** ppSetList = NULL;

    // Validate
    if ( !m_pAnimController || !m_pAnimController->GetNumAnimationSets( ) )
        return D3DERR_INVALIDCALL;

    // Store set count
    ULONG SetCount = m_pAnimController->GetNumAnimationSets( );

    // Allocate array to hold animation sets
    ppSetList = new ID3DXAnimationSet*[ SetCount ];
    if ( !ppSetList ) return E_OUTOFMEMORY;

    // Clear the array
    ZeroMemory( ppSetList, SetCount * sizeof(ID3DXAnimationSet*) );
```

We now loop and fetch each animation set from the animation controller and store the retrieved base interface pointer in our local animation set array. As we copy over each animation set interface, we release that animation set from the controller so that the only reference to the animation set interface is now in our array.

```
    // Loop through each animation set and copy into our set list array
    for ( i = 0; i < SetCount; ++i )
    {
        // Keep retrieving index 0 (remember we unregister each time)
        hRet = m_pAnimController->GetAnimationSet( 0, &pAnimSet );
        if ( FAILED( hRet ) ) continue;

        // Store this set in our list, and AddRef
        ppSetList[ i ] = pAnimSet;
        pAnimSet->AddRef();
```

```
        // Unregister the old animation sets first otherwise we won't have enough
        // 'slots' for the new ones.
        m_pAnimController->UnregisterAnimationSet( pAnimSet );

        // Release this set, leaving only the item in the set list.
        pAnimSet->Release();

    } // Next Set
```

The second pass is where we copy over the data from the original animation set into a new CAnimationSet. We do this by allocating a new CAnimationSet object and passing into the constructor the ID3DXAnimationSet base interface of the set we intend to clone. The constructor will take care of copying over all the keyframe data and storing it in the CAnimationSet's internal variables (if it is an ID3DXKeyframedAnimationSet that is passed in). Because this copying is happening in the constructor we have no way of returning success or failure, so we use exceptions. If an exception is thrown from inside the constructor, the catch block will simply delete the CAnimationSet we just allocated. When this happens, the original animation set's pointer is left in our array and will be re-registered with the controller later on.

```
    // Loop through each animation set and rebuild a new custom one.
    for ( i = 0; i < SetCount; ++i )
    {
        // Skip if we didn't retrieve anything
        if ( ppSetList[i] == NULL ) continue;

        // Allocate a new animation set, duplicating the one we just retreived
        // Note : Because this is done in the constructor, we catch the
        // exceptions because no return value is available.
        try
        {
            // Duplicate
            pNewAnimSet = new CAnimationSet( ppSetList[i] );

        } // End Try Block
        catch ( HRESULT & e )
        {
            // Release the new animation set
            delete pNewAnimSet;

            // Just skip this animation set (leave it in the array)
            continue;

        } // End Catch block
        catch ( ... )
        {
            // Just skip this animation set (leave it in the array)
            continue;

        } // Catch all other errors (inc. out of memory)
```

If we get this far without incident, then the animation set has been successfully copied into a new CAnimationSet object. The original animation set can then be released and a pointer to our new CAnimationSet stored in the array in its place.

```
        // Release this set, memory will now be freed
        ppSetList[i]->Release();

        // Store the new animation set in our temporary array
        ppSetList[i] = pNewAnimSet;

    } // Next Set
```

Finally, in the third pass we loop through each animation set in the array and register it with the controller. We then delete the temporary array and bind the first animation set to the first track of the controller.

```
    // third pass, register all new animation sets
    for ( i = 0; i < SetCount; ++i )
    {
        // Register our brand new set. (AddRef is called)
        m_pAnimController->RegisterAnimationSet( ppSetList[i] );

        // Release our hold on the custom animation set
        ppSetList[i]->Release();

    } // Next Set

    // Free up any temporary memory
    delete []ppSetList;

    // Set the first animation set into the first track, otherwise
    // nothing will play, since all previous tracks have been unregistered.
    SetTrackAnimationSetByIndex( 0, 0 );

    // We're all done
    return D3D_OK;

}
```

When this function returns back to LoadActorFromX, all ID3DXKeyframedAnimationSets will have been replaced with our CAnimationSets.

We have now covered all the new methods involved with loading and initializing the actor for animation. The remaining functions exposed by the actor are used by the application to play animations and configure the playback settings.

CActor::AdvanceTime

The new AdvanceTime method is the heartbeat of our actor. The application will call this method periodically to update the animation system's internal clock. This is really just a wrapper function around a call to the ID3DXAnimationController::AdvanceTime method, which updates the animation system and generates the new relative frame matrices for the hierarchy.

The first thing you will often want to do after updating the frame matrices is traverse the hierarchy and build the absolute matrices for each frame. You will recall from the previous lab project that we did this, during each iteration of our game loop, using the CActor::UpdateFrameMatrices method. This method traverses the frame hierarchy and combines all relative matrices to generate the world matrices for each frame in the hierarchy. It makes sense that we would want to perform this task *after* animating the local frame matrices because this step would invalidate the previous world matrices.

For convenience, we have added a Boolean parameter called UpdateFrames to this function (set to true by default). When set to true, the function will automatically call CActor::UpdateFrameMatrices before returning. This allows the actor to update the animation and rebuild the absolute matrices for each frame with a single call to CActor::AdvanceTime. The UpdateFrameMatrices method is unchanged from the previous lab project so will not be covered again here.

There may be times when you do not wish the world matrices of each frame to be calculated immediately after an animation update has taken place. In this case you can pass false as this parameter. You might do this because the application intends to make some manual adjustments to the relative frame matrices prior to the world matrix rebuilding. If the application does pass false to the UpdateFrames parameter, it must make sure that it calls CActor::UpdateFrameMatrices prior to rendering the actor. If it does not, the animations applied to the relative frame matrices will not be reflected in their world space counterparts used for the mesh transformations.

The CActor::AdvanceTime method is shown below.

```
void CActor::AdvanceTime(     double fTimeElapsed,
                              bool UpdateFrames /* = true */,
                              LPD3DXANIMATIONCALLBACKHANDLER pCallbackHandler)
{
    if ( !IsLoaded() ) return;

    // Set the current time if applicable
    if ( m_pAnimController )
        m_pAnimController->AdvanceTime( fTimeElapsed, pCallbackHandler );

    // Update the frame matrices
    if ( UpdateFrames ) UpdateFrameMatrices( m_pFrameRoot, &m_mtxWorld );
```

The application passes the amount of elapsed time since the previous call to the function, which is then routed to the ID3DXAnimationController::AdvanceTime method. As discussed earlier, when the ID3DXAnimationController::AdvanceTime method returns program flow back to our function, the relative hierarchy frame matrices will have been updated to represent their new positions. Finally, we call the UpdateFrameMatrices method which uses the updated relative matrices to build the

absolute/world matrices for each frame that will be used for transformation and rendering of any meshes in the hierarchy.

As the third parameter to the CActor::AdvanceTime method, the application can pass a pointer to a ID3DXAnimationCallbackHandler derived class. This base abstract interface exposes only one function which we must implement if we wish our callback keys to be used. When the controller executes a callback key, it will call the ID3DXAnimationCallbackHandler::HandleCallback method. Obviously this will only happen if we have passed into the controller the pointer to an object that is derived from this interface.

If you take a look in the d3dx9anim.h header file which ships with the SDK, you can see that the ID3DXAnimationCallbackHandler interface is declared as:

```
DECLARE_INTERFACE(ID3DXAnimationCallbackHandler)
{
    STDMETHOD(HandleCallback)(THIS_ UINT Track, LPVOID pCallbackData) PURE;
};
```

This is a pure virtual base class rather than an interface; it is not even derived from IUnknown. What does this mean for us? Well, all we have to do is derive our own class from this base class and implement its single method: HandleCallback. We can then pass a pointer to our object into the CActor::AdvanceTime method, where it will passed to the ID3DXAnimationController::AdvanceTime method. From there, the controller will call its HandleCallback method whenever a callback key is executed.

When this method is called by the controller, it is passed two parameters. The first is the track number for which the callback key was triggered. Your callback function might use this to determine if it wishes to take any action with respect to the current track. The second parameter is the callback key context data pointer. Remember, this will contain either NULL or the address of a CActionData object. Recall from our earlier discussion that our actor enforces the rule that callback key context pointers should point to an IUnknown derived class so that the actor can clean it up properly. In our application, we use the CActionData object for storing data that we would like passed to our callback method. In Lab Project 10.1, each CActionData object will contain the name of a wave file sound effect that the callback function can play when the key is triggered.

If you look in CScene.h you will see that we have derived our own class from ID3DXAnimationCallbackHandler like so:

```
class CActionHandler : public ID3DXAnimationCallbackHandler
{
public:
     STDMETHOD(HandleCallback)(THIS_ UINT Track, LPVOID pCallbackData);
};
```

Note that we have added nothing to this class and used a straight implementation of the interface laid out by the base class. It is a pointer to a CActionHandler class that we will pass into the

CActor::AdvanceTime method. Its HandleCallback method will be called by the controller whenever a callback key needs to be processed.

The CActionHandler::HandleCallback method is implemented in CScene.cpp (see below). Our callback function is very simple and does not use the track number passed by the controller for any purpose. It is only interested in the second parameter, which contains a pointer to our CActionData object (containing the name of a sound file that will need to be played).

The first thing we do is cast the void context data to an IUnknown interface. We know that whatever object may be pointed at by this void pointer has to be either NULL or one derived from IUnknown since our CActor class insists on it. Once we have cast it to IUnknown, we had better make sure that this is a CActionData object since it is what our callback handler is interested in handling. Therefore, we call IUnknown::QueryInterface to check that it is indeed one of our CActionData objects. If it is not, then we return immediately since this callback function does not know how to handle any other data type. If it is a CActionData object, then we are returned a CActionData interface which we can then use to extract the wave filename.

```
HRESULT CActionHandler::HandleCallback( UINT Track, LPVOID pCallbackData )
{
    IUnknown    * pData   = (IUnknown*)pCallbackData;
    CActionData * pAction = NULL;

    // Determine if this is in a format we're aware of
    if ( FAILED( pData->QueryInterface( IID_CActionData, (void**)&pAction ) ) )
         return D3D_OK; // Just return unhandled.
```

Now that we have an interface to our CActionData object, we check its m_Action member variable. This can currently only be set to ACTION_NONE or ACTION_PLAYSOUND, but you may decide to add different types of actions to this class yourself. Our callback function only handles sound effect callback keys at the moment, so we have to make sure that we only work with CActionData objects that are in ACTION_PLAYSOUND mode.

```
    // This is our type, lets handle it
    switch ( pAction->m_Action )
    {
        case CActionData::ACTION_PLAYSOUND:

            // Play the sound if available
            if ( pAction->m_pString != NULL )
                 sndPlaySound( pAction->m_pString, SND_ASYNC | SND_NODEFAULT );
            break;

    } // End Switch Action Type

    // Release the action we queried.
    pAction->Release();

    // We're all done
    return D3D_OK;
}
```

If this is an ACTION_PLAYSOUND mode CActionData object, we fetch the filename from its m_pString member and pass it into a Win32 function (sndPlaySound) which loads and plays the specified wave file. We then release the interface to the CActionData object and return from the function.

In our lab project we register two callback keys of type ACTION_PLAYSOUND. Thus, this callback function will be called twice by the controller in a given loop of animation. Since the animation is set to loop, these same two callback keys will cause this function to be called over and over again so that the sounds play during every loop of the animation.

sndPlaySound

To keep things simple, we are using the Win32 function sndPlaySound to play our sound effects. This function takes two parameters. The first is the name of the wave file we wish to play and as you can see, we pass in the name of the wave file stored inside the CActionData object. The second parameter is a combination of one or more of the following flags:

Value	Meaning
SND_ASYNC	The sound is played asynchronously and the function returns immediately after beginning to play the sound. To terminate an asynchronously played sound, call **sndPlaySound** with *lpszSoundName* set to NULL. We use this option so that we can start a long sound playing and continue with our application processing.
SND_LOOP	The sound plays repeatedly until **sndPlaySound** is called again with the *lpszSoundName* parameter set to NULL. You must also specify the SND_ASYNC flag to loop sounds.
SND_MEMORY	If this flag is specified, then the first parameter should not point to a string containing the filename of the wave file but should instead point to an image of a waveform sound in memory.
SND_NODEFAULT	If the sound cannot be found, the function returns silently without playing the default sound. We use this option so that in the event that the sound file is missing, we do not end up playing some default sound instead that might be out of context with our scene.
SND_NOSTOP	If a sound is currently playing, the function immediately returns FALSE, without playing the requested sound. If you specify this flag and a sound is already playing, no action will be taken. Without this flag, the currently playing sound would be cut short and the new sound played immediately.
SND_SYNC	The sound is played synchronously and the function does not return until the sound ends. We do not want to use this flag because our animation would freeze and our application would receive no frame updates until the sound had finished playing.

This is certainly not a function that you would use in a commercial game project (that is what DirectXSound is for), especially since it can only play one sound at a time. However, the purpose of this exercise is to demonstrate the callback mechanism, so this works nicely for us.

Note: In order to use the sndPlaySound function, you must make sure that you are linking to the winmm.lib library (part of the platform SDK). This library includes all the Win32 multimedia functionality.

Advancing our Actor

Let us look at an example of how the actor is updated in our application. You should be fairly comfortable by now with our application framework and the fact that we usually handle any scene based animation updates in the CScene::AnimateObjects method. This method is called for every iteration of our game loop by the CGameApp::FrameAdvance method, giving the scene a chance to update the positions of any objects prior to rendering. The function is passed the application CTimer object so that it can retrieve the elapsed time since the last frame update and forward this time to the CActor::AdvanceTime method.

Note: This is not the exact CScene::AnimateObjects call found in Lab Project 10.1, but it is very close. We have to discuss some additional topics before we are ready to look at the full version of this function. However, we will see all but a few lines of the code and show the important points for this discussion.

```
void CScene::AnimateObjects( CTimer & Timer )
{
    static CActionHandler Handler;

    // Process each object
    for ( ULONG i = 0; i < m_nObjectCount; ++i )
    {
        // Increase actor's reference time if this stores an actor
        if ( m_pObject[i]->m_pActor )
         m_pObject[i]->m_pActor->AdvanceTime( Timer.GetTimeElapsed(),
                                              false,
                                              &Handler );

    } // Next Object
}
```

The function starts by declaring a variable of type CActionHandler. This object contains the callback function that we wish the controller to use to process callback keys. Since we will need to use this object every frame, we make it a static variable rather than a stack variable that has to be repeatedly created and destroyed. We did not make this a member of the CScene class because the only time this object is ever used is in this function. So we decided to keep its scope safely limited to this function. By making it static we can have the best of both worlds -- a global variable that is created once with a safe scope limited to this one function.

We then loop through each CObject in the scene's object array. You will recall from the previous lab project that our scene may consist of single mesh objects and objects with actors attached and still behave properly. In this function however, we are only interested in objects which are actors. So for each object that has an actor attached, we call the actor's AdvanceTime method, passing in the elapsed time retrieved from the timer along with a pointer to our callback handler. At the end of this loop, each actor in the scene will have had its frame hierarchy animated. In Lab Project 10.1 we only have a single actor which will contain all of the objects in the scene and their associated animations.

Why are we passing in false as the second parameter? Recall from our earlier discussion that if it is set to true, then the actor will automatically traverse the hierarchy and build the absolute matrices for each frame. You might imagine that this is surely something we want to have done, right?

While it is true that we need to update the absolute matrices after animation has been applied, we certainly do not wish to perform multiple traversals if we can avoid it. You will recall from the previous lab project that before rendering an actor we must also set its world matrix so that we can position it in the world at an arbitrary location and orientation. (The application can move and position the actor in the scene by applying transformations to the matrix stored in the parent CObject to which the actor is attached.) In the CScene::Render method, prior to rendering the actor's subsets, we call CActor::SetWorldMatrix and pass in the parent CObject matrix as the root node matrix:

```
pActor->SetWorldMatrix( &m_pObject[i]->m_mtxWorld, true );
```

As we learned in the previous workbook, this function is passed the world matrix of the actor as its first parameter and it caches this matrix internally. It is by manipulating the CObject::m_mtxWorld matrix that the application can move the entire actor about in the scene. If true is passed as the second parameter, then the passed matrix is combined with the root node matrix and all changes are filtered down through the hierarchy. Since this call happens in the render function (after the AnimateObjects method), it would be wasteful for us to pass true to the CActor::AdvanceTime method. The result would be a redundant and pointless hierarchy traversal. If we passed true to CActor::AdvanceTime, the hierarchy frames would be generated and the absolute world matrices for each frame constructed and stored. However, when CScene::Render calls CActor::SetWorldMatrix, the absolute matrices we just generated would be obsolete because we have just re-positioned the root node of the actor. Thus, all absolute frame matrices would need to be rebuilt again. Many of the actor methods expose this UpdateFrames Boolean option, but you only want to pass true to the final one called before rendering.

To summarize, using our actor class requires the following steps (some of which are optional):

1. Create a CActor object.
2. Use CActor::RegisterCallback method to register any callback functions for callback key registration and texture and material parsing. (OPTIONAL).
3. Set the desired maximum extents of the actor using the CActor::SetActorLimits method. (OPTIONAL)
4. Load the X file into the actor using CActor::LoadActorFromX.
5. During every iteration of your game loop:
    a. Increment the actor's internal clock using CActor::AdvanceTime (will update the actor's frame matrices).
    b. Set the world matrix for the actor using CActor::SetWorldMatrix.
    c. Render the actor by looping through each subset and calling CActor::DrawSubset.

CActor Utility Functions

We have already discussed the core functions of the actor and saw how easy it is to operate via its interface. However, the actor must also expose many utility functions to allow the application to get and set the properties of the underlying animation controller. For example, the application will want the ability to assign different animation sets to different tracks and to control the global time of the controller. The application will also want to be able to register sequencer events and change the properties of a mixer track to control how animation sets are blended. These are all methods exposed by the ID3DXAnimationController interface which are now (for the most part) wrapped by CActor. Let us take a look at them now.

CActor::ResetTime

CActor::ResetTime can be called to reset the global clock of the animation controller to zero. The reason an application might want to do this is related to the fact that with each call to CActor::AdvanceTime, the global time of the controller gets higher and higher. If the application was left running for a very long time, this global time would eventually reach a very high number. If left long enough, it might loop round to zero by itself, causing jumps in the animation. Furthermore, as sequencer events are registered on the global timeline, this is our only means for returning the global clock of the controller back to zero so that those sequencer events can be triggered again.

This function wraps the ID3DXAnimationController::ResetTime method. The reset time method is a little more complicated than it might first seem. Not only does it reset the global timer of the controller back to zero, but it also wraps around each of the track timers. It does not simply set them to zero since this would cause jumps in the animation. Instead it wraps them around such that the track timer is as close to zero as possible without altering the periodic position of the animation set currently being played. This function is designed to reset the global time to 0.0 while retaining the current periodic position for all playing tracks.

```
void CActor::ResetTime( bool UpdateFrames /* = false */ )
{
    if ( !IsLoaded() ) return;

    // Set the current time if applicable
    if ( m_pAnimController ) m_pAnimController->ResetTime( );

    // Update the frame matrices
    if ( UpdateFrames ) UpdateFrameMatrices( m_pFrameRoot, &m_mtxWorld );
}
```

Notice how this function also has the UpdateFrames Boolean so that the application can force an immediate rebuild of the hierarchy's absolute frame matrices. For example, if this is the last CActor method you are calling prior to rendering the actor's subset you might decide to do this. Normally, you will pass false as this parameter (the default).

CActor::GetTime

This method wraps the ID3DXAnimationController::GetTime method. It returns the current accumulated global time of the animation controller.

```
double CActor::GetTime( ) const
{
    if ( !m_pAnimController) return 0.0f;
    return m_pAnimController->GetTime();
}
```

The GetTime method is useful if an animation wants to perform its own looping logic. For example, imagine that a given animation plays out for 25 seconds and that the track the animation set is assigned to has several sequencer events registered within that time frame. Because sequencer events are on the global timeline, even if the animation set was configured to loop, each sequencer event would only be triggered once. Once the global time of the controller increased past 25 seconds, it would continue to grow and the events would never be triggered again. However, if the application wants these sequencer events to loop with the animation set, it could retrieve the global time of the actor using CActor::GetTime and, if it has climbed over a certain threshold, reset back to zero. This would cause the time to reset every 25 seconds in our example. Thus, the animation set would loop and the sequencer events on the global timeline would be triggered each time.

CActor::GetMaxNumTracks

This function returns the number of tracks currently available on the animation controller. It is a wrapper around the ID3DXAnimationController::GetMaxNumTracks method. For example, the application might use this method to query the current number of tracks available on the mixer and if not enough, the application could call the CActor::SetActorLimits method to extend this numer.

```
ULONG CActor::GetMaxNumTracks( ) const
{
    if ( !m_pAnimController ) return 0;
    return m_pAnimController->GetMaxNumTracks();
}
```

Assigning Animation Sets to Mixer Tracks

The application will often need the ability to assign different animation sets to tracks on the animation mixer. For example, imagine the actor is a game character. When the actor is in a walking state, the application would probably just assign a walking animation set to a mixer track and play that track on its own. But if the character were to suddenly fire its weapon, the application would also want to assign a firing animation set to another track and blend the two animations together so that the character appears to continue walking and fire his weapon. The ID3DXAnimationController interface exposes a single method for this purpose called ID3DXAnimationController::SetTrackAnimationSet. This function

accepts a track number and a pointer to the animation set interface you wish to assign to that track. This must be an animation set that has been previously registered with the animation controller.

Setting a track in this way can sometimes seem a rather clunky approach. Often, our application may not have a pointer to the interface of the animation set, which means it must be retrieved from the controller first (performing a search by name or by index). It would be much nicer if our actor also allowed the application to specify the name of an animation set or its index as well. For example, if we know that we have an animation set called 'Fire' which we wish to assign to track 1, we would first have to retrieve an interface pointer by calling ID3DXAnimationSet::GetAnimationSetByName, and then pass the returned animation set interface into the SetTrackAnimationSet interface. To remove this burden from the application, our CActor actually exposes three methods (SetTrackAnimationSet, SetTrackAnimationSetByName, and SetTrackAnimationSetByIndex) that can be used to assign animation sets to mixer tracks in a variety of different ways.

These methods allow the application to bind an animation set to a mixer track using either its interface, its numerical index within the controller or by using the name of the animation set. They are shown below with a brief discussion of each.

CActor::SetTrackAnimationSet

This method is a wrapper around its ID3DXAnimationController counterpart. The application must pass two parameters; the first should be the zero based index of the mixer track that the specified animation set should be assigned to and the second should be the ID3DXAnimationSet interface pointer we wish to set.

```
HRESULT CActor::SetTrackAnimationSet(ULONG TrackIndex,
                                     LPD3DXANIMATIONSET pAnimSet )
{
    if ( !m_pAnimController ) return D3DERR_INVALIDCALL;

    // Set anim set
    return m_pAnimController->SetTrackAnimationSet( TrackIndex, pAnimSet );
}
```

This function forwards the parameters to the ID3DXAnimationController::SetTrackAnimationSet method. In order to use this method, the application must already have an interface to the animation set it wishes to assign to a track. We can use the CActor::GetAnimationSet method (discussed in a moment) for this purpose.

CActor::SetTrackAnimationSetByIndex

We implemented this function to make the assignment of animation sets to mixer tracks a one line call within the application code. If the application knows the zero based index of the animation set it wishes to assign, then this function is the one to use. The application passes the track to assign the animation set to and the index of the animation set.

```
HRESULT CActor::SetTrackAnimationSetByIndex( ULONG TrackIndex, ULONG SetIndex )
{
    HRESULT hRet;
    LPD3DXANIMATIONSET pAnimSet = NULL;

    if ( !m_pAnimController ) return D3DERR_INVALIDCALL;

    // Retrieve animation set by index
    hRet = m_pAnimController->GetAnimationSet( SetIndex, &pAnimSet );
    if ( FAILED(hRet) ) return hRet;

    // Set anim set
    hRet = m_pAnimController->SetTrackAnimationSet( TrackIndex, pAnimSet );

    // Release the one we retrieved
    pAnimSet->Release();

    // Return result
    return hRet;
}
```

This method automates some of the work the application would usually have to do. It calls the ID3DXAnimationController::GetAnimationSet method to retrieve an interface to the animation set at the specified index. As discussed in the textbook, this method accepts an index and the address of an ID3DXAnimationSet interface pointer. If a valid animation set exists in the controller at the specified index, its interface will be returned in the second parameter. We then pass this interface into the ID3DXAnimationController::SetTrackAnimationSet method, along with the track index, to perform the actual track assignment. Notice that we release the interface that was returned from ID3DXAnimationController::GetAnimationSet so the object's reference count is correctly maintained.

CActor::SetTrackAnimationSetByName

This final track setting method is almost identical to the previous function in way that it behaves. This time, the function should be passed a track index and a string containing the name of the animation set we wish to assign to the specified mixer track.

The function uses the ID3DXAnimationController::GetAnimationSetByName method, which is passed the name of the animation set and the address of an ID3DXAnimationSet interface pointer. If an animation set with the specified name exists within the animation controller, on function return, the pointer passed as the second parameter will point to the interface of this animation set. Once the animation set interface has been retrieved, we can bind it to the desired track using the ID3DXAnimationController::SetTrackAnimationSet method as we have done in the previous two methods.

```
HRESULT CActor::SetTrackAnimationSetByName( ULONG TrackIndex, LPCTSTR SetName )
{
    HRESULT hRet;
    LPD3DXANIMATIONSET pAnimSet = NULL;
```

```
    if ( !m_pAnimController ) return D3DERR_INVALIDCALL;

    // Retrieve animation set by name
    hRet = m_pAnimController->GetAnimationSetByName( SetName, &pAnimSet );
    if ( FAILED(hRet) ) return hRet;

    // Set anim set
    hRet = m_pAnimController->SetTrackAnimationSet( TrackIndex, pAnimSet );

    // Release the one we retrieved
    pAnimSet->Release();

    // Return result
    return hRet;
}
```

Using CActor to Alter Mixer Track Properties

As discussed in the textbook, the animation controller will have an internal multi-track animation mixer available for use. Much like an audio mixing desk, each track has a number of properties that can be altered to change the way the animation set is played and combined into the final result. We can enable or disable a track midway through an animation, alter the track position to adjust the timeline of a single animation set, change the speed at which the track position is incremented with respect to the global timer, and change the weight of a track so that the SRT data contributes more or less to the final matrix generated for that frame.

The ID3DXAnimationController provides methods to allow our application to perform all of these tasks, and so too will our CActor. In the next section, we will look at a series of small wrapper functions around the corresponding animation controller methods that manage the animation mixer properties and behavior.

CActor::SetTrackPosition

This method allows the application to alter the timer of a single track on the mixer. We pass in the index of the track we wish to alter and the new position we would like to set the track's timer to. Remember that this is track time (not periodic animation set time) that we are setting here. Since the track position is mapped to a periodic position, altering the track position allows us to manually jump, skip, or pause an animation. For example, if we called this function in our game loop passing in the same track position each time, the animation would appear frozen. This is because the same track position would be mapped to the same periodic position resulting in identical SRT data. Of course, if the animation set was also being blended with other tracks, only its contribution would be frozen. The other tracks would continue to play out as expected.

```
HRESULT CActor::SetTrackPosition( ULONG TrackIndex, DOUBLE Position )
{
    if ( !m_pAnimController ) return D3DERR_INVALIDCALL;
```

```
    // Set the position within the track.
    return m_pAnimController->SetTrackPosition( TrackIndex, Position );
}
```

CActor::SetTrackEnable

This function is passed a track index and a Boolean variable. If the Boolean is set to true, the track will be enabled and any animation set assigned to that track will play out as expected. The track will remain enabled until it is disabled. If false is passed, the specified track will be disabled and will remain so until it is re-enabled by the application. When a track is disabled, it will be ignored by the animation controller. Any animation set assigned to that track will not contribute to the frame hierarchy the next time AdvanceTime is called.

```
HRESULT CActor::SetTrackEnable( ULONG TrackIndex, BOOL Enable )
{
    // Validate
    if ( !m_pAnimController ) return D3DERR_INVALIDCALL;

    // Set the details
    return m_pAnimController->SetTrackEnable( TrackIndex, Enable );
}
```

CActor::SetTrackPriority

In the textbook we learned that each track on the mixer can be assigned to one of two priority blend groups. The fact that these groups are referred to as high and low priority is perhaps a bit misleading as the blend ratio between these two groups can be altered such that the low priority group could contribute more to the frame hierarchy than the high priority group.

This method allows us to assign a track to one of the two priority groups. We specify the priority group using the D3DXPRIORITY_TYPE enumeration shown below:

```
typedef enum _D3DXPRIORITY_TYPE
{
    D3DXPRIORITY_LOW = 0,
    D3DXPRIORITY_HIGH = 1,
    D3DXEDT_FORCE_DWORD = 0x7fffffff
} D3DXPRIORITY_TYPE;
```

```
HRESULT CActor::SetTrackPriority( ULONG TrackIndex, D3DXPRIORITY_TYPE Priority )
{
    // Validate
    if ( !m_pAnimController ) return D3DERR_INVALIDCALL;

    // Set the details
    return m_pAnimController->SetTrackPriority( TrackIndex, Priority );
}
```

CActor::SetTrackSpeed

We discussed in the textbook that each track has its own independant position and its own speed (set to 1.0 by default). Using this method we can speed up/slow down the rate at which a single animation set is playing without affecting animation sets assigned to any other tracks on the mixer. The application passes this function the index of the track and the new speed.

```
HRESULT CActor::SetTrackSpeed( ULONG TrackIndex, float Speed )
{
    // Validate
    if ( !m_pAnimController ) return D3DERR_INVALIDCALL;

    // Set the details
    return m_pAnimController->SetTrackSpeed( TrackIndex, Speed );
}
```

The track speed is changed internally by the controller. It uses the track speed property to scale the elapsed time passed into the AdvanceTime method before adding the result to the track position. As a simple example, let us imagine we pass in an elapsed time of 5 seconds. Five seconds would be added to the global clock of the animation controller. Now the controller needs to add this elapsed time to the position of each track to update the track timelines also. However, before it does this, it scales elapsed time by the track speed. If we imagine that the track speed was set to 3.0 and that the current track position was 20.0, the track position would be updated like so:

TrackPosition =20
ElapsedTime = 5
Track Speed = 3.0

NewTrackPosition = TrackPosition + (ElapsedTime * TrackSpeed)
= 20 + (5 * 3)
= 20 + 15
= 35

If we imagine that another currently active mixer track also has a track position of 20 but only has a track speed of 0.5 we can see that the track position for this track would be updated much less during thr same advance time call:

TrackPosition = 20
ElapsedTime = 5
Track Speed = 0.5

NewTrackPosition = TrackPosition + (ElapsedTime * TrackSpeed)
= 20 + (5 * 0.5)
= 20 + 2.5
= 22.5

As you can see, the first animation set will appear to play much quicker than the second one in a single update. In the first case, with an elapsed time of 5 seconds, the timeline of the track was actually updated 15 seconds to a new track position of 35 seconds. The second animation set was advanced only by 2.5 seconds to a new track position of 22.5 seconds.

CActor::SetTrackWeight

The weight of a track controls how strongly the SRT data generated by its assigned animation set contributes to the overall hierarchy. It also controls the weight at which it is blended with other animation sets/tracks assigned to the same priority group.

```
HRESULT CActor::SetTrackWeight( ULONG TrackIndex, float Weight )
{
    // Validate
    if ( !m_pAnimController ) return D3DERR_INVALIDCALL;

    // Set the details
    return m_pAnimController->SetTrackWeight( TrackIndex, Weight );
}
```

The default weight for each track is 1.0. This means that the SRT data generated by the animations of the animation set will not be scaled and will influence the hierarchy at full strength. If the weight of track A is set to 0.5 and the weight assign to track B is 1.0 and they both belong to the same priority group, the animation set assigned to track B will influence the hierarchy more strongly than track A. Setting track weights is very important as it allows you to find the right balance when multiple animations are being blended and played. We will see many examples of multi-track animation blending as we progress through the course.

CActor::SetTrackDesc

The actor also exposes a function that allows the application to set all the track properties with a single function call. This function is a wrapper around ID3DXAnimationController::SetTrackDesc. It accepts a track index and a pointer to a D3DXTRACK_DESC structure which contains the new properties for the track

```
HRESULT CActor::SetTrackDesc( ULONG TrackIndex, D3DXTRACK_DESC * pDesc )
{
    // Validate
    if ( !m_pAnimController ) return D3DERR_INVALIDCALL;

    // Set the details
    return m_pAnimController->SetTrackDesc( TrackIndex, pDesc );
}
```

Here we see the D3DXTRACK_DESC structure as a reminder:

```
typedef struct _D3DXTRACK_DESC {
    D3DXPRIORITY_TYPE Priority;
    FLOAT Weight;
    FLOAT Speed;
    DOUBLE Position;
    BOOL Enable;
} D3DXTRACK_DESC, *LPD3DXTRACK_DESC;
```

Retrieving the Properties of a Mixer Track

Just as an application using our actor will need the ability to set track properties on the animation mixer, it will also need a means for retrieving the current values being used by those mixer tracks. The ID3DXAnimationController exposes two methods that allow the application to retrieve the properties of the mixer.

CActor::GetTrackDesc

CActor::GetTrackDesc is the exact opposite of the method previously discussed. The application passes in the track number it wishes to retrieve properties for and the address of a D3DXTRACK_DESC structure. On function return, this structure will be filled with the properties of the specified mixer track.

```
HRESULT CActor::GetTrackDesc( ULONG TrackIndex, D3DXTRACK_DESC * pDesc ) const
{
    // Validate
    if ( !m_pAnimController ) return D3DERR_INVALIDCALL;

    // Get the details
    return m_pAnimController->GetTrackDesc( TrackIndex, pDesc );
}
```

CActor::GetTrackAnimationSet

The second method exposed by the animation controller to communicate track properties to the application is ID3DXAnimationController::GetTrackAnimationSet. This allows an application to retrieve an interface to the animation set currently assigned to a mixer track.

This function is passed only a track index and returns a pointer to an ID3DXAnimationSet interface for that animation set assigned to the track.

```
LPD3DXANIMATIONSET CActor::GetTrackAnimationSet( ULONG TrackIndex ) const
{
    LPD3DXANIMATIONSET pAnimSet;

    // Validate
    if ( !m_pAnimController ) return NULL;

    // Get the track details (calls AddRef) and return it
    if ( FAILED(m_pAnimController->GetTrackAnimationSet( TrackIndex, &pAnimSet )))
        return NULL;
    return pAnimSet;
}
```

Setting the Priorty Blend Weight of the Controller

The animation controller has a global property for all tracks called the priority blend weight. It is a single scalar value (usually between 0.0 and 1.0) which determines how the high and low priority track groups on the mixer will be combined during the final blending stage. Setting this value to 0.5 for example, will blend the SRT data generated from both blend groups using a 50/50 mix.

Our actor wraps the ID3DXAnimationController methods for setting and retrieving this blend weight. CActor::SetPriorityBlend and CActor::GetPriortyBlend are two methods exposed to the application for this purpose:

```
HRESULT CActor::SetPriorityBlend( float BlendWeight )
{
    // Validate
    if ( !m_pAnimController ) return D3DERR_INVALIDCALL;

    // Set the details
    return m_pAnimController->SetPriorityBlend( BlendWeight );

}
```

```
float CActor::GetPriorityBlend( ) const
{
    // Validate
    if ( !m_pAnimController ) return 0.0f;

    // Get the details
    return m_pAnimController->GetPriorityBlend( );
}
```

Querying the Controller's Animation Sets

The ID3DXAnimationController interface exposes methods to allow the application to fetch interfaces for registered animation sets. Unlike the ID3DXAnimationController::GetTrackAnimationSet method, we can use these methods to fetch interfaces for animation sets that are not currently assigned to any track on the animation mixer. Obviously we need to be able to fetch animation sets which are not currently assigned to mixer tracks so that we can assign them to tracks on the mixer. The controller also exposes methods that allow the application to query the number of animation sets currently registered with the controller as well as the maximum number supported.

This collection of functionality is covered by four different actor methods. The first method allows the application to query the maximum number of animation sets that can be registered with the animation controller simultaneously:

```
ULONG CActor::GetMaxNumAnimationSets( ) const
{
    if ( !m_pAnimController ) return 0;
    return m_pAnimController->GetMaxNumAnimationSets( );
}
```

The second method allows the application to query the number of animation sets that are currently registered with the animation controller:

```
ULONG CActor::GetAnimationSetCount( ) const
{
    // Validate
    if ( !m_pAnimController ) return 0;

    // Return the count
    return m_pAnimController->GetNumAnimationSets( );
}
```

The next two methods allow us to fetch interfaces to animation sets that are currently registered with the controller. The first method allows us to fetch the animation set interface using an index. This is the zero based index of the animation set within the controllers's array of registered animation sets:

```
LPD3DXANIMATIONSET CActor::GetAnimationSet( ULONG Index ) const
{
    LPD3DXANIMATIONSET pAnimSet;

    // Validate
    if ( !m_pAnimController ) return 0;

    // Get the animation set and return it
    if ( FAILED( m_pAnimController->GetAnimationSet( Index, &pAnimSet ) ))
        return NULL;
    return pAnimSet;
}
```

Your application may not always know the index of an animation set it wishes to fetch, but may know the name of the animation. This second function allows us to pass in a string containing the name of the animation set we wish to retrieve an interface for. As with the above function, this one wraps the call to its ID3DXAnimationController counterpart. If an animation set is registered with the animation controller which matches the name passed into the function, an ID3DXAnimationSet interface to that animation set will be returned:

```
LPD3DXANIMATIONSET CActor::GetAnimationSetByName( LPCTSTR strName ) const
{
    LPD3DXANIMATIONSET pAnimSet;

    // Validate
    if ( !m_pAnimController ) return 0;

    // Get the animation set and return it
    if ( FAILED( m_pAnimController->GetAnimationSetByName( strName, &pAnimSet ) ))
```

```
        return NULL;
    return pAnimSet;
}
```

Registering Track Events with the Animation Sequencer

As discussed in the textbook, the animation controller includes an event sequencer that allows an application to schedule track property changes for a specific global time. Just as each property of a mixer track can be set with a call to one of the SetTrackXX methods discussed ealier, each track property also has a method in the ID3DXAnimationController for scheduling that track property change to occur at a specific time. Our CActor has wrapped these methods and the source code to these functions will be dicussed in this section. Since these are simply wrapper functions around the methods of the animation controller (discussed in the main course text), these functions will requite little explanation.

> **Note:** When using these sequencer functions, remember that events are specified in global time. That is, the global time of the controller is incremented with each AdvanceTime call. Global time does not loop unless the timer is specifically reset using CActor::ResetTime. Also remember that the global time of the controller may be very different from the time of a specific track. This will be true if the track position has been manipulated by the application (see CActor::SetTrackPosition) or if the speed of a track is not the default. If the global time is never reset and continues to increment, a registered sequencer event will only ever occur once. This is true even if the animation set assigned to that track is configured to loop.

CActor::KeyTrackEnable

This method can be used by the application to schedule a track to be enabled or disabled at a specific global time. The application should pass in the track index as the first parameter and the second parameter should be set to true or false indicating whether the track should be enabled or disabled by this event. The third parameter is the global time at which this event should occur.

```
D3DXEVENTHANDLE CActor::KeyTrackEnable( ULONG TrackIndex, BOOL NewEnable,
                                        double StartTime )
{
    // Validate
    if ( !m_pAnimController ) return NULL;

    // Send the details
    return m_pAnimController->KeyTrackEnable( TrackIndex, NewEnable, StartTime );
}
```

Our method simply forwards the request on to the animation controller. This method is a sequencer version of the CActor::SetTrackEnable method discussed previously.

CActor::KeyTrackPosition

This method allows the application to schedule a track position manipulation at a specified global time. Essentially it allow us to manipulate track time at a specified global time. It is important that we distinguish between the two different timelines being used here. As the first paramater the track index is passed for which the position will be changed. As the second parameter we pass in the new track position that we would like this track to be set to at the time when this event is triggered.

The 'NewPosition' is specified in track time, not global time and not as a periodic position. We might for example set this NewPosition to 20 seconds for example. If the animation had a period of 10 seconds, then this would essentially cause the animation to jump to the beginning of its 3rd iteration (if looping is enabled for this animation set). This is because a track position of 20 seconds would be mapped to a periodic position of 0 seconds for a looping animation with a 10 second period.

The third parameter is the time at which we would like this track position change event to occur. Unlike the second parameter, this is not specified in track time but is instead specified in global time which might be quite different. As mentioned, the sequencer events are registered on the global time line.

```
D3DXEVENTHANDLE CActor::KeyTrackPosition( ULONG TrackIndex, double NewPosition,
                                          double StartTime )
{
    // Validate
    if ( !m_pAnimController ) return NULL;

    // Send the details
    return m_pAnimController->KeyTrackPosition(TrackIndex,NewPosition,StartTime);
}
```

This method is the sequencer version of the CActor::SetTrackPosition method discussed previously.

CActor::KeyTrackSpeed

This method allows the application to schedule a track speed property change to occur at a specified global time. We pass in the track index and the new speed we would like this track to be set to when the event is triggered. The third parameter is the global time at which we would like this event to be triggered. The fourth parameter is the event duration (in seconds). The duration describes how many seconds we would like it to take, after the event has been triggered, to transition the current speed to the new speed. This feature (along with the next parameter) allows us to schedule smoother transitions.

```
D3DXEVENTHANDLE CActor::KeyTrackSpeed( ULONG TrackIndex, float NewSpeed,
                                       double StartTime, double Duration,
                                       D3DXTRANSITION_TYPE Transition )
{
    if ( !m_pAnimController ) return NULL;

    // Send the details
    return m_pAnimController->KeyTrackSpeed( TrackIndex, NewSpeed,
                                             StartTime, Duration, Transition );
}
```

The fifth parameter is a member of the D3DXTRANSITION_TYPE enumeration which describes how we would like the change from the current speed to the new speed to occur over the specified duration. The two choices are linear interpolation and spline based interpolation.

```
typedef enum _D3DXTRANSITION_TYPE {
    D3DXTRANSITION_LINEAR = 0x000,
    D3DXTRANSITION_EASEINEASEOUT = 0x001,
    D3DXEDT_FORCE_DWORD = 0x7fffffff
} D3DXTRANSITION_TYPE;
```

Whichever transition type we use, the change from the old speed to the new speed will still be performed within the specified duration. If linear is being used then the current speed will linearly ramp up/down from the old speed to the new speed over the duration. If ease-in ease-out is used, the speed change will start off slowly, gradually increase until some maximum, level off, and then, when near the end of the specified duration, the speed change will decrease until the final target is reached. As the member of the enumerated type suggests, this changes the speed in a more natural way, speeding up and slowing down at the outer ends of the duration.

This method is the sequencer version of the SetTrackSpeed method discussed previously.

CActor::KeyTrackWeight

This method allows the application to schedule a weight change for a specified track at a specified global time. By sequencing weight change events, we can essentially script the strength at which an animation set influences the frame hierarchy over a period of time.

```
D3DXEVENTHANDLE CActor::KeyTrackWeight( ULONG TrackIndex, float NewWeight,
                                        double StartTime, double Duration,
                                        D3DXTRANSITION_TYPE Transition )
{
    // Validate
    if ( !m_pAnimController ) return NULL;

    // Send the details
    return m_pAnimController->KeyTrackWeight( TrackIndex, NewWeight, StartTime,
                                              Duration, Transition );
}
```

This function, like the previous one, also accepts a duration and transition type parameter. This allows the weight of the track to be changed from its current weight (at the time the event is triggered) to its new weight, over a specified time in seconds (the Duration parameter). The transition type once again controls whether we wish the change from the current weight to the new weight of the track to be interpolated linearly or using a spline based approach.

This method is the sequencer version of the CActor::SetTrackWeight method discussed previously.

Registering Priorty Blend Changes with the Sequencer

The animation controller has a global property called the *priority blend*. This value is usually set between zero and one and is used to weight the contributions of the high and low priority groups that are blended together in the final mixing stage.

As discussed in the textbook, mixer tracks can be assigned to one of two priority groups (high or low). When AdvanceTime is called, the SRT data for each animation set is generated. Once we have the SRT data generated from each track, all SRT data belong to tracks in the same priority groups are blended together using their track weights to weight their contribution in the mix. This is done for both the high and low priority groups. At this point, the animation controller has two groups of SRT data: the output from the low priority mixing phase and the output from the high priority mixing phase. Once this is output is prepared, the SRT data in these two groups is blended together. The priority blend value of the controller is used to weight the contribution of the two groups in the final mix. The output of this final stage is a single set of SRT data used to rebuild the relative matrix of the attached animated frame.

```
D3DXEVENTHANDLE CActor::KeyPriorityBlend( float NewBlendWeight, double StartTime,
                                          double Duration,
                                            D3DXTRANSITION_TYPE Transition )
{
    // Validate
    if ( !m_pAnimController ) return NULL;

    // Send the details
    return m_pAnimController->KeyPriorityBlend( NewBlendWeight,
                                                StartTime, Duration, Transition );
}
```

Just like the track speed and weight changes, when we schedule an event to change the priority blend weight, we can also specify a duration and transition type. This allows us to slowly change the priority blend weight from its old value to the new one.

This method is the sequencer version of the CActor::SetPriorityBlend method discussed previously.

Removing Sequencer Events from the Global Timeline

There may be times when you need to un-register a sequencer event. Perhaps your application has reset the global time of the controller but does not wish a certain event to be triggered again the second time through. This next series of methods allow the application to unregister sequencer events in a variety of different ways.

CActor::UnkeyEvent

The ID3DXAnimationController has an UnkeyEvent method that we have wrapped in CActor::UnkeyEvent. The method is passed the event handle of the event you wish to unregister. All

event registration functions return a D3DXEVENTHANDLE, which is a unique ID for that sequencer event. If you wish to perform some action on that event later (e.g., remove it), this is the handle that you can use to identify this specific event to the D3DX animation system.

If the passed D3DXEVENTHANDLE matches the handle of an event currently registered on the global timeline, this method will remove it. This method can be used to unregister all event types (weight, speed, position, etc). The animation controller will automatically know which type of event it is and which track that event has to be removed from based on the passed event handle.

```
HRESULT CActor::UnkeyEvent( D3DXEVENTHANDLE hEvent )
{
    // Validate
    if ( !m_pAnimController ) return D3DERR_INVALIDCALL;

    // Send the details
    return m_pAnimController->UnkeyEvent( hEvent );
}
```

CActor::UnkeyAllTrackEvents

This method removes all events that currently exist for a given track. The application simply passes in the index of the track and the controller will remove any events that have been registered for that track.

```
HRESULT CActor::UnkeyAllTrackEvents( ULONG TrackIndex )
{
    // Validate
    if ( !m_pAnimController ) return D3DERR_INVALIDCALL;

    // Send the details
    return m_pAnimController->UnkeyAllTrackEvents( TrackIndex );
}
```

Events that have been registered for other mixer tracks are not affected by this method. Note as well that priority blend events are *not* removed by this method. This is because priority blend events are not specific to a given track but instead alter the priority blend weight of the animation controller, which is used by all tracks. Any tracks events that were previously registered with the track index passed into the function will no longer be valid when the function returns.

CActor::UnkeyAllPriorityBlends

Priority blend events are unlike all other event types because they are not track specific. Instead, they alter a global property of the controller. Because of this, the CActor::UnkeyAllTrackEvents method cannot be used to unregister events of this type. The CActor::UnkeyAllPriorityBlends method should be used to clear all priority blend events from the global timeline.

```
HRESULT CActor::UnkeyAllPriorityBlends( )
{
    // Validate
```

```
    if ( !m_pAnimController ) return D3DERR_INVALIDCALL;

    // Send the details
    return m_pAnimController->UnkeyAllPriorityBlends( );
}
```

Validating Events

The CActor::ValidateEvent function can be used to test if an event is still valid. Valid events are events which have not yet been executed (because the global time is smaller than the timestamp of the event) or events that are currently being executed. The method accepts a single parameter, the D3DXEVENTHANDLE of the event you wish to validate. Any event whose timestamp is less than the global time of the controller is considered invalid.

```
HRESULT CActor::ValidateEvent( D3DXEVENTHANDLE hEvent )
{
    // Validate
    if ( !m_pAnimController ) return D3DERR_INVALIDCALL;

    // Send the details
    return m_pAnimController->ValidateEvent( hEvent );
}
```

When the global time of the controller is reset, all events that were previously considered invalid become valid again. The exception of course, is if you pass in the handle of an event that your application has since un-registered.

Fetching the Details for an Upcoming Event

It is often useful for an application to know when a sequencer event is currently being executed so that the application can perform some action when the event is triggered. This is much like how an application might respond to a callback function. However, a callback method is not used for this purpose. Instead, the application can poll for an event type on a specific track or poll for a currently executing priority blend event using this next series of functions. If an event is currently being executed that fits the specified criteria, the handle of the event will be returned. The CActor::GetEventDesc method can then be used to fetch the event details of this event using the passed event handle.

CActor::GetCurrentTrackEvent

This method is used to test if an event is currently being executed on a given track. The function accepts two parameters. For the first parameter, the application should pass the index of the track for which it is requesting event information. For the second parameter, the application should pass a member of the D3DXEVENT_TYPE enumeration specifying the type of events it is interested in being informed about.

This method just wraps the corresponding ID3DXAnimationController method. If an event of the specified type is currently being executed on the specified track, the handle of the event will be returned from the method.

```
D3DXEVENTHANDLE CActor::GetCurrentTrackEvent( ULONG TrackIndex,
                                              D3DXEVENT_TYPE EventType ) const
{
    // Validate
    if ( !m_pAnimController ) return NULL;

    // Send the details
    return m_pAnimController->GetCurrentTrackEvent( TrackIndex, EventType );
}
```

The D3DXEVENT_TYPE enumeration is defined as follows in the DirectX header files.

```
typedef enum _D3DXEVENT_TYPE {
    D3DXEVENT_TRACKSPEED = 0,
    D3DXEVENT_TRACKWEIGHT = 1,
    D3DXEVENT_TRACKPOSITION = 2,
    D3DXEVENT_TRACKENABLE = 3,
    D3DXEVENT_PRIORITYBLEND = 4,
    D3DXEDT_FORCE_DWORD = 0x7fffffff
} D3DXEVENT_TYPE;
```

Passing D3DXEVENT_PRIORITYBLEND into this method will never return a valid handle since priority blend events are not track events.

CActor::GetCurrentPriorityBlend

This method accepts no parameters and will return the handle of the currently executing priority blend event. If no priority blend event is being executed then the function will return NULL.

```
D3DXEVENTHANDLE CActor::GetCurrentPriorityBlend( ) const
{
    // Validate
    if ( !m_pAnimController ) return NULL;

    // Send the details
    return m_pAnimController->GetCurrentPriorityBlend( );
}
```

CActor::GetEventDesc

This method can be used to fetch the details of any sequencer event registered on the global timeline. It accepts two parameters. The first parameter is the handle of the event for which information is to be retrieved (via parameter two). This could be the handle returned from either the CActor::GetCurrentPriorityBlend or CActor::GetCurrentTrackEvent if you wish to retrieve information

about an event that is currently being executed. The method simply wraps its ID3DXAnimationController counterpart.

```
HRESULT CActor::GetEventDesc( D3DXEVENTHANDLE hEvent, LPD3DXEVENT_DESC pDesc )
const
{
    // Validate
    if ( !m_pAnimController ) return D3DERR_INVALIDCALL;

    // Send the details
    return m_pAnimController->GetEventDesc( hEvent, pDesc );
}
```

CActor Utility Functions

The final two methods of the CActor interface fall into the category of utility functions. These are methods that you are not likely to use very often when integrating CActor into your day-to-day projects, but they do come in handy when using the CActor class as part of a tool. The first method is simply an upgrade from Lab Project 9.1.

CActor::SaveActorToX

This method is used to save the actor out to an X file. Unlike the previous version of this function, we now pass the D3DXSaveMeshHierarchyToFile function the address of the animation controller interface. This will ensure that D3DX will save the frame and mesh data to the X file as well as the animation data. This single change to this function is highlighted in bold in the listing below.

```
HRESULT CActor::SaveActorToX( LPCTSTR FileName, ULONG Format )
{
    HRESULT hRet;

    // Validate parameters
    if ( !FileName ) return D3DERR_INVALIDCALL;

    // If we are NOT managing our own attributes, fail
    if ( GetCallback( CActor::CALLBACK_ATTRIBUTEID ).pFunction != NULL )
         return D3DERR_INVALIDCALL;

    // Save the hierarchy back out to file
    hRet = D3DXSaveMeshHierarchyToFile( FileName, Format,
                                        m_pFrameRoot, m_pAnimController,
                                        NULL );
    if ( FAILED(hRet) ) return hRet;

    // Success!!
    return D3D_OK;
}
```

CActor::ApplySplitDefinitions

This next function is a large one, but the code does not use any techniques we have not yet encountered. Before examining the code, let us first discuss the service this method is intended to provide and why an application would be interested in such a service.

As discussed in the textbook, some X file exporters only support saving animated X files with a single animation set. The Deep Exploration™ plugin for 3D Studio Max™ supports multiple animation set export and is a really excellent X file exporter. If you can afford both 3D Studio and this plugin, you would be hard pressed to find a better and easier solution. However, most students just getting started in their game programming studies simply cannot afford such professional tools. So a workaround for this problem is often needed.

One way to get around this problem requires artist participation. The artist can store all the animation sequences for a given object directly after each other on a single timeline. For example, if an artist was building animation data for a game character, he might use the first 10 seconds of the timeline to store the walking animation, the next 10 seconds to store the idle animation, the next 10 seconds for the death animation, and so on. If this character's animation was played back in this arrangement, it would essentially cycle through all possible animations that the character is capable of as the timeline is traversed from start to finish.

In Lab Project 10.2 we use an X file called 'Boney.x' that has been constructed in this way. Because all the animation sequences are stored on the same timeline and are essentially part of the same animation set (as far as the modeling application is concerned), this X file can be exported by applications that support saving only a single animation set. All the animation data we need will be saved in the X file.

The problem is that all of these sequences will exist in a single animation set and this is far from ideal. Normally, we would like these animations to be stored in separate animation sets so that each individual sequence can be played back by the animation controller in isolation. We would also like the ability to blend, for example, a walking animation sequence with a firing animation sequence. We know that in order to accomplish this, we must have those two sequences in different animation sets so that they can be assigned to different tracks on the animation mixer. But having been supplied with this X file which contains all the animation data in a single set, what are we to do?

Our answer was to implement a new method for CActor called ApplySplitDefinitions, which allows us to correct this problem after the X file has been loaded. This method will allow the application to pass in an array of split definition structures. Each structure will describe how we would like a section of the current 'single' animation set to be copied into a new animation set. For example, if we had a single animation set with five sequences stored in it, this function could be used to break it into five separate animation sets. Each split definition structure that we pass in would contain the start and end positions on the original 'combined' timeline where the new animation set will start and end. It would also contain the name that we would like to call this new animation set. Once the method returns, the original animation set would have been released and we would now have five separate animation sets in its place.

This approach does require that the artist let us know the start and end positions of each sequence in the set so that our code can correctly fill in the split definition structures before passing them to the ApplySplitDefinitions function. However, this can also prove to be a limitation, especially if the artist is not available for us to question. So we have designed something that negates the need for even that requirement.

In Lab Project 10.2 we will develop the source code for a tool called The Animation Splitter. It is a simple dialog box driven GUI tool wrapped around the CActor::ApplySplitDefinitions method. The Splitter will allow us to graphically set markers on the combined timeline and apply the splits into separate sets. Behind the scenes this tool will call the CActor::ApplySplitDefinitions method to split the animation set into multiple animation sets based on the markers you set via the user interface. Then you can save the actor back out to an X file with its newly separated animation sets.

Whether you use this tool to split up your combined animations as a pre-process or use the CActor::ApplySplitDefinitions method in your main code after loading the X file is up to you. Bear in mind that after you have applied the splits, you can always use the CActor::SaveActorToX method to save the data back out to file in its new form. This GUI tool is just a user-friendly way to configure the call to CActor::ApplySplitDefinitions. So let us have a look at this final CActor method which will be used by the animation splitter in the next project to do all the heavy lifting.
The CActor::ApplySplitDefinitions method accepts an array of AnimSplitDefinition structures This structure (defined in CActor.h) is shown below.

```
typedef struct _AnimSplitDefinition // Split definition structure
{
    TCHAR   SetName[128];       // The name of the new animation set
    ULONG   SourceSet;          // The set from which we're sourcing the data
    double  StartTicks;         // Ticks at which this new set starts
    double  EndTicks;           // Ticks at which this new set ends
    LPVOID  Reserved;           // Reserved for use internally, do not use.

} AnimSplitDefinition;
```

Each stucture of this type that we pass into the function will describe a new animation set that we wish to create from a segment of an existing one. The members are described below:

TCHAR SetName[128]
This member is where supply the name of the new animation set we wish to be created.

ULONG SourceSet
In this member we supply the index of the animation set currently registered with the controller that we would like the data for this new animation set to be extracted from. In the case of a single combined animation set, this will always be set to zero since the controller will only have one set. However, the function will handle splitting from multiple sources. This is essentially the source information for this definition. It is the set which will have a section of its keyframe data copied into a new animation set. This set will be released after the splits have been applied.

double StartTicks
This is the starting marker on the source set's animation timeline indicating where the section of animation data we wish to copy into a new set begins. For example, if we were supplied a single animation set and informed that the walking sequence for this animation began at tick 400 and ended at tick 800, we would set both the StartTicks and EndTicks members of this structure to 400 and 800 respectively. Any animation data in the source animation set between these two tick positions will be copied into the new set.

double EndTicks
This member describes the end marker for the section of animation data we wish to copy into the new animation set from the source animation set.

LPVOID Reserved
This member is reserved and should not be used.

Let us first look at an example of how an application might use the CActor::ApplySplitDefinitions function, before we cover the code. Continuing our previously discussed example, let us imagine we have loaded an X file into our actor which contains two animation sequences that we would like to divide into two separate animations sets called 'Walk' and 'Run' respectively. Let us also imagine that we are informed by our artist that the walking sequence is contained in the animation timeline between ticks 0 and 400 and the running sequence is contained in the original animation set between ticks 400 and 800. To perform this action, we would write the following code:

```
// Allocate an array for two definitions
AnimSplitDefinition    AnimDefs[2];
UINT                   NumberOfAnimDefs=2;

// Set up definition 1 called 'Walk'
_tcscpy( AnimDefs[0].SetName, _T("Walk") );
AnimDefs[0].SourceSet   =     0
AnimDefs[0].StartTicks  =     0
AnimDefs[0].EndTicks    =     400

// Set up defnition 2 called 'Run'
_tcscpy( AnimDefs[1].SetName, _T("Run") );
AnimDefs[0].SourceSet   =     0
AnimDefs[0].StartTicks  =     400
AnimDefs[0].EndTicks    =     800

// Create the two new animation sets and release the original ( set 0 )
pActor->ApplySplitDefinitions( AnimDefsCount , AnimDefs );
```

That is all there is to the process. The animation controller of the actor now has two separate animation sets called 'Walk' and 'Run' where previously only a single animation set (set 0) exisited. The original animation set will have been unregistered from the controller and freed from memory when the CActor::ApplySplitDefinitions function returns.

Let us now study the code. This is going to be quite a large function so we will need to cover it a step at a time. Fortunately, there is hardly anything here that we have not studied and implemented already, so for the most part it will be easy going.

The first section of the function has the job of cloning the actor's animation controller if it cannot support the number of animation sets passed in the nDefCount parameter. Remember, each split definition in this array is describing a new animation set we would like to create and register with the animation controller. If the animation controller is currently limited to only having two animation sets registered with it, but the application has passed in an array of 10 split definitions, we know we must have an animation controller that is capable of having the 10 new animation sets we are about to create registered with it.

```
HRESULT CActor::ApplySplitDefinitions( ULONG nDefCount,
                                       AnimSplitDefinition pDefList[] )
{
    HRESULT                     hRet;
    LPCTSTR                     strName;
    ULONG                       i, j, k;
    LPD3DXANIMATIONSET          pAnimSet;
    LPD3DXKEYFRAMEDANIMATIONSET pSourceAnimSet, pDestAnimSet;

    D3DXKEY_VECTOR3            *ScaleKeys     = NULL, *NewScaleKeys = NULL;
    D3DXKEY_VECTOR3            *TranslateKeys = NULL, *NewTranslateKeys = NULL;
    D3DXKEY_QUATERNION         *RotateKeys    = NULL, *NewRotateKeys = NULL;

    ULONG                       ScaleKeyCount, RotationKeyCount,
                                TranslationKeyCount, AnimationCount;

    ULONG                       NewScaleKeyCount,
                                NewRotationKeyCount,
                                NewTranslationKeyCount;

    D3DXVECTOR3                 StartScale, EndScale,
                                StartTranslate, EndTranslate;

    D3DXQUATERNION              StartRotate, EndRotate;

    // Validate pre-requisites
    if ( !m_pAnimController || !IsLoaded() || nDefCount == 0 )
        return D3DERR_INVALIDCALL;

    // Clone our animation controller if there are not enough set slots available
    if ( m_pAnimController->GetMaxNumAnimationSets() < nDefCount )
    {
        LPD3DXANIMATIONCONTROLLER pNewController = NULL;

        // Clone the animation controller
        m_pAnimController->CloneAnimationController(
                                m_pAnimController->GetMaxNumAnimationOutputs(),
                                nDefCount,
                                m_pAnimController->GetMaxNumTracks(),
                                m_pAnimController->GetMaxNumEvents(),
                               &pNewController );
```

```
        // Release our old controller
        m_pAnimController->Release();

        // Store the new controller
        m_pAnimController = pNewController;

    } // End if too small!
```

Notice in the above code that when we clone the animation controller we feed in all the same controller limits as the current controller, with the exception of the second parameter. This is where we pass in the number of animation sets we will need this new controller to handle. Therefore, the new controller will inherit all the limits of the original controller but with an increase in the number of animation sets that can be registered. The original controller is then released and the actor's controller member pointer assigned the address of our cloned controller.

We will now set up a loop to process each split definition in the passed array. For each one, we will fetch the index of the source set it references. This is the index of the animation set from which the keyframe data will be extracted. If this is not an ID3DXKeyframedAnimationSet then we will skip this definition. Provided it is a keyframed animation set, we create a new ID3DXKeyframedAnimationSet. The name of this set is the name that was specified by the application and passed in the split definition.

```
    // We're now going to start building for each definition
    for ( i = 0; i < nDefCount; ++i )
    {
        // First retrieve the animation set
        if ( FAILED(m_pAnimController->GetAnimationSet( pDefList[i].SourceSet,
                                                        &pAnimSet )) ) continue;

        // We have to query this to determine if it's keyframed
        if ( FAILED( pAnimSet->QueryInterface( IID_ID3DXKeyframedAnimationSet,
                                               (LPVOID*)&pSourceAnimSet ) ) )
        {
            // Release animation set and continue
            pAnimSet->Release();
            continue;

        } // End if failed to retrieve keyframed set

        // We can release the original animation set now, we've 'queried' it
        pAnimSet->Release();

        // Create a new 'destination' keyframed animation set
        D3DXCreateKeyframedAnimationSet(pDefList[i].SetName,
                                        pSourceAnimSet->GetSourceTicksPerSecond(),
                                        pSourceAnimSet->GetPlaybackType(),
                                        pSourceAnimSet->GetNumAnimations(),
                                        0,
                                        NULL,
                                        &pDestAnimSet );
```

Look at the call to the D3DXCreateKeyframedAnimationSet function above. With the exception of the first parameter, where we pass in a new name, we inherit all other properties from the source set (playback type, number of animations, and source ticks per second ratio).

With our new animation set created, our next task is to loop through each animation in the source animation set and extract the scale, rotation, and translation keys into temporary arrays. The following code fetches the SRT key counts for the animation currently being processed. It then allocates SRT arrays to store a copy of the source SRT data. Then, it stores a copy of the SRT key data in the temporary SRT arrays so that we have a complete copy of the keyframe data to work with. Notice that one of the first things we do is fetch the name of the animation we are processing. We will need this later when we wish to register the copy of the animation with the new animation set. We need to make sure that the animation names remain intact since this provides the mapping between the animation data and the frame in the hierarchy it animates.

```
        // Loop through the animations stored in the set
        AnimationCount = pSourceAnimSet->GetNumAnimations();
        for ( j = 0; j < AnimationCount; ++j )
        {
            // Get name
            pSourceAnimSet->GetAnimationNameByIndex( j, &strName );

            // Retrieve all the key counts etc
            ScaleKeyCount           = pSourceAnimSet->GetNumScaleKeys( j );
            RotationKeyCount        = pSourceAnimSet->GetNumRotationKeys( j );
            TranslationKeyCount     = pSourceAnimSet->GetNumTranslationKeys( j );
            NewScaleKeyCount        = 0;
            NewRotationKeyCount     = 0;
            NewTranslationKeyCount  = 0;

            // Allocate enough memory for the keys
            if ( ScaleKeyCount )
                ScaleKeys = new D3DXKEY_VECTOR3[ ScaleKeyCount ];

            if ( TranslationKeyCount )
                TranslateKeys = new D3DXKEY_VECTOR3[ TranslationKeyCount ];

           if ( RotationKeyCount )
                RotateKeys = new D3DXKEY_QUATERNION[ RotationKeyCount ];

            // Allocate enough memory (total) for our new keys,
            // + 2 for potential start and end keys
            if ( ScaleKeyCount )
                  NewScaleKeys = new D3DXKEY_VECTOR3[ ScaleKeyCount + 2 ];

           if ( TranslationKeyCount )
                  NewTranslateKeys = new D3DXKEY_VECTOR3[TranslationKeyCount+ 2 ];

           if ( RotationKeyCount )
                  NewRotateKeys = new D3DXKEY_QUATERNION[ RotationKeyCount + 2 ];

            // Retrieve the physical keys
            if ( ScaleKeyCount )
               pSourceAnimSet->GetScaleKeys( j, ScaleKeys );
```

```
            if ( TranslationKeyCount )
               pSourceAnimSet->GetTranslationKeys( j, TranslateKeys );

            if ( RotationKeyCount )
                 pSourceAnimSet->GetRotationKeys( j, RotateKeys );
```

The inner loop is concerned with copying the keyframe data for a single animation in the source set into a single animation in the destination set. Further, we will only copy over keyframes that fall within the start and end markers specified in the definition. What we must consider however is that the definition may have a start or end marker set between two keyframes in the source set. For example, imagine an animation has two translation keys at 0 and 10 seconds respectively. It is possible that we have placed a start and end marker at 3 and 8 seconds, respectively. Since the only two keyframes that exist fall outside these two markers, we would end up not copying anything into the new animation set. This is certainly not the way it should work.

In such a case, if the start and end markers do not fall on pre-existing keyframe positions, we must generate them ourselves and insert them into the new animation set. In the current example, this means we would need to generate a start and an end keyframe to be inserted at 3 and 8 seconds and. These are the two keyframes that the final animation would contain.

So it seems that we might have a situation where the start or end marker is not situated on pre-existing keyframe positions, and when this is the case we may need to insert a new start or end keyframe (or both, if both markers are between existing keyframes). But how do we generate a scale, rotation and translation key for these ‘in-between’ times?

When you think about this problem, you realize that we are asking a question we have already answered numerous times before. The ID3DXKeyframedAnimationSet::GetSRT method does just that. All we have to do is pass it the start marker position and it will interpolate the correct scale, rotation and translation keys for those in-between times. In the next section of code, we call the GetSRT method twice to generate SRT keys for the StartTicks and EndTicks positions specified in the current definition. Because the GetSRT method expects a time in seconds and we have our split definition marker values in ticks, we must pass in the Start and End marker positions divided by the source ticks per second value of the animation set.

```
            // Get the SRT data at the start and end ticks
            // (just in case we need to insert them)
            pSourceAnimSet->GetSRT( (double)pDefList[i].StartTicks /
                                     pSourceAnimSet->GetSourceTicksPerSecond(), j,
                                    &StartScale, &StartRotate, &StartTranslate );

            pSourceAnimSet->GetSRT( (double)pDefList[i].EndTicks /
                                     pSourceAnimSet->GetSourceTicksPerSecond(), j,
                                    &EndScale, &EndRotate, &EndTranslate );

            // Swap 'winding', we're adding these directly backto the anim
            // controller, not using directly.
            D3DXQuaternionConjugate( &StartRotate, &StartRotate );
            D3DXQuaternionConjugate( &EndRotate, &EndRotate );
```

We may not actually need the two sets of SRT keys we have just generated. If the start and end markers fall on existing keyframe positions then we can simply copy them over into the new animation. However, if this is not the case, then we require the above step to interpolate the keys for the start and end markers. Notice that we swap the handedness of the quaternions returned so they are left-handed.

Now it is time to start copying over the relevant data from the source animation into the new animation. We will do this in three passes.

In this first pass we copy over the scale keys. In this first section of the scale key pass, we loop through each scale key in the array and skip any keys which have timestamps prior to the StartTicks position specified in the current split definition being processed. If a key has a smaller timestamp we simply continue the next iteration of the loop to process the next scale key. If the timestamp of the current scale key is larger than the EndTicks time (also supplied in the split definition structure) then it means that we have processed all scale keys between the start and end marker positions and we are not interesting in copying any keys after this point. When this is the case, we break from the loop and our job is done.

```
            // ******************************************************************
            // * SCALE KEYS
            // ******************************************************************
            for ( k = 0; k < ScaleKeyCount; ++k )
            {
                D3DXKEY_VECTOR3 * pScale = &ScaleKeys[k];

                // Skip all keys prior to the start time
                if ( pScale->Time < pDefList[i].StartTicks ) continue;

                // Have we hit a key past our last time?
                if ( pScale->Time > pDefList[i].EndTicks ) break;
```

If we get here then we have a key that needs to be copied. However, if we have not yet added any scale keys then this is the first. Also, if the timestamp of this first key we are about to add is not exactly the same as the StartTicks marker position stored in the definition, it means that the first keyframe we are about to copy occurs after the start tick position. We need a keyframe to be inserted at the start marker position so we insert the start scale key we calculated above using the GetSRT method. This will make this start key the first in the array (now definitively located at the start marker position). We then copy over the current key we are processing.

```
                // If we got here we're within range.
                //If this is the first key, we may need to add one
                if(NewScaleKeyCount==0 && pScale->Time != pDefList[i].StartTicks )
                {
                    // Insert the interpolated start key
                    NewScaleKeys[ NewScaleKeyCount ].Time  = 0.0f;
                    NewScaleKeys[ NewScaleKeyCount ].Value = StartScale;
                    NewScaleKeyCount++;

                } // End if insert of interpolated key is required

                // Copy over the key and subtract start time
                NewScaleKeys[ NewScaleKeyCount ] = *pScale;
```

```
            NewScaleKeys[ NewScaleKeyCount ].Time -= pDefList[i].StartTicks;
            NewScaleKeyCount++;
        } // Next Key
```

At the end of this loop we will have copied over all scale keys that existed in the original animation between the start and end marker positions. Further, we may have inserted a new start key at the beginning of the array if the first key we intended to copy had a greater timestamp than the starting marker position. We must always have keyframes on the start and end marker positions.

Now we must test to see if the last scale key we added in the above loop has a timestamp equal to the EndTicks position. If so, then all is well because the last key we copied is exactly where this animation set's timeline should end. However, if this is not the case, then it means the last scale key we copied occurred before the EndTicks position. In that case, we must insert an additional key at the end marker position. We calculated that end key ealier using GetSRT, so we add it to the end of the array.

```
        // Last key matched end time?
        if ( NewScaleKeys[NewScaleKeyCount - 1].Time !=
             pDefList[i].EndTicks - pDefList[i].StartTicks )
        {
            // Insert the interpolated end key
            NewScaleKeys[ NewScaleKeyCount ].Time  = pDefList[i].EndTicks -
                                                     pDefList[i].StartTicks;
            NewScaleKeys[ NewScaleKeyCount ].Value = EndScale;
            NewScaleKeyCount++;

        } // End if insert of interpolated key is required
```

That is the end of the first pass. We now have a temporary array called NewScaleKeys which contains all scaling information between the two markers specified by the split definition for the current animation being copied.

In the second pass we will do exactly the same thing, only this time we will be copying over translation keys instead. The code is nearly identical.

```
        // ******************************************************************
        // * TRANSLATION KEYS
        // ******************************************************************
        for ( k = 0; k < TranslationKeyCount; ++k )
        {
            D3DXKEY_VECTOR3 * pTranslate = &TranslateKeys[k];

            // Skip all keys prior to the start time
            if ( pTranslate->Time < pDefList[i].StartTicks ) continue;

            // Have we hit a key past our last time?
            if ( pTranslate->Time > pDefList[i].EndTicks ) break;

            // If we got here we're within range.
            // If this is the first key, we may need to add one
            if (NewTranslationKeyCount==0 &&
               pTranslate->Time!= pDefList[i].StartTicks )
```

```
                {
                    // Insert the interpolated start key
                   NewTranslateKeys[NewTranslationKeyCount].Time  = 0.0f;
                   NewTranslateKeys[NewTranslationKeyCount].Value =
                                                              StartTranslate;
                   NewTranslationKeyCount++;
                } // End if insert of interpolated key is required

                // Copy over the key and subtract start time
              NewTranslateKeys[NewTranslationKeyCount] = *pTranslate;
              NewTranslateKeys[NewTranslationKeyCount].Time -=
                                                    pDefList[i].StartTicks;
              NewTranslationKeyCount++;

            } // Next Key

            // Last key matched end time?
            if ( NewTranslateKeys[NewTranslationKeyCount - 1].Time !=
                 pDefList[i].EndTicks - pDefList[i].StartTicks )
            {
                // Insert the interpolated end key
                NewTranslateKeys[ NewTranslationKeyCount ].Time =
                              pDefList[i].EndTicks - pDefList[i].StartTicks;

              NewTranslateKeys[ NewTranslationKeyCount ].Value = EndTranslate;
                NewTranslationKeyCount++;

            } // End if insert of interpolated key is required
```

Once again, the code shows that if there are no keyframes existing in the source animation set on the start and end marker positions, we insert the start and end keys interpolated using GetSRT prior to the copy process.

At this point, two of the three copy passes are complete. We have two arrays that contain scale keys and translation keys that existed either on or between the two markers. In the final pass, we copy over the rotation information using the same strategy. Once again, we insert the start and end quaternion keys we interpolated earlier if it is necessary to use them.

```
            // ******************************************************************
            // * ROTATION KEYS
            // ******************************************************************
            for ( k = 0; k < RotationKeyCount; ++k )
            {
                D3DXKEY_QUATERNION * pRotate = &RotateKeys[k];

                // Skip all keys prior to the start time
                if ( pRotate->Time < pDefList[i].StartTicks ) continue;

                // Have we hit a key past our last time?
                if ( pRotate->Time > pDefList[i].EndTicks ) break;

                if ( NewRotationKeyCount == 0 &&
                   pRotate->Time != pDefList[i].StartTicks )
                {
```

```
                // Insert the interpolated start key
                NewRotateKeys[ NewRotationKeyCount ].Time  = 0.0f;
                NewRotateKeys[ NewRotationKeyCount ].Value = StartRotate;
                NewRotationKeyCount++;

            } // End if insert of interpolated key is required

            // Copy over the key and subtract start time
            NewRotateKeys[ NewRotationKeyCount ] = *pRotate;
            NewRotateKeys[ NewRotationKeyCount ].Time -=
                                                pDefList[i].StartTicks;
            NewRotationKeyCount++;

        } // Next Key

        // Last key matched end time?
        if ( NewRotateKeys[NewRotationKeyCount - 1].Time !=
           pDefList[i].EndTicks - pDefList[i].StartTicks )
        {
            // Insert the interpolated end key
            NewRotateKeys[ NewRotationKeyCount ].Time  =
                                pDefList[i].EndTicks - pDefList[i].StartTicks;

            NewRotateKeys[ NewRotationKeyCount ].Value = EndRotate;
            NewRotationKeyCount++;

        } // End if insert of interpolated key is required
```

With the copy process complete, we now have three arrays that describe the SRT data for one of the animations in our newly created animation set. Next we will register that SRT data with our new animation set. Notice that we supply the same name for the animation we are adding to the new animation set as the animation we are copying from. We retrieved this information previously at the start of the animation loop. We must make sure that when we register this new animation copy with our new animation set that we give it the same name as the animation it was copied from. This name provides the mapping between the SRT data contained in the animation and the hierarchy frame which it animates. Once we have added the new animation to our new animation set, we can release the temporary arrays we allocated for the copy process.

```
        // Register this with our destination set
        pDestAnimSet->RegisterAnimationSRTKeys(strName,
                                               NewScaleKeyCount,
                                               NewRotationKeyCount,
                                               NewTranslationKeyCount,
                                               NewScaleKeys,
                                               NewRotateKeys,
                                               NewTranslateKeys, NULL );

        // Clean up
        if ( ScaleKeys )   delete []ScaleKeys; ScaleKeys = NULL;
        if ( TranslateKeys )    delete []TranslateKeys; TranslateKeys = NULL;
        if ( RotateKeys )       delete []RotateKeys; RotateKeys = NULL;
        if ( NewScaleKeys )     delete []NewScaleKeys; NewScaleKeys = NULL;
```

```
            if ( NewRotateKeys )    delete []NewRotateKeys; NewRotateKeys = NULL;
            if ( NewTranslateKeys )
               delete []NewTranslateKeys; NewTranslateKeys = NULL;

        } // Next Animation
```

The animation copy loop we have just examined is executed for every animation in the source set. This creates corresponding (albeit edited) versions of those same animations in the new set. Once we have exited the inner loop, we have copied over all required animation data and have successfully dealt with the current split definition being processed. We have created a new animation set for it and have populated it with updated animation data.

Before moving on to processing the next split definition in the array, we release the interface to the original animation set. This does not delete it from memory since the controller is currently still using it. Our new animation sets will not be added to the controller until later, so for the time being, the controller still contains the original sets. In fact, the last thing we would want to do at this point is actually remove the source animation set from memory. Remember, we may have more split definitions to process and some of them may (and probably will) use the same source animation set for the copying process. So we will still need access to it. This will certainly be the case when the X file contains a single animation set containing all the animation sequences. We are only releasing the copy of the interface pointer that we were returned from pAnimationController->GetAnimationSet( ) at the start of the split definitions loop. The outer loop (i.e., the split definition loop) closes like so:

```
        // We're done with the source 'keyframed' animation set
        pSourceAnimSet->Release();

        // Store the dest animation set for use later
        pDefList[i].Reserved = (LPVOID)pDestAnimSet;

    } // Next Definition
```

Notice that this function uses the reserved member of the split definition structure it has just processed to store a temporary pointer to the new animation set it just created. Now we see why this member was reserved -- it gives the function a temporary place to store the animation sets created for each split definition until we register them with the controller later on.

At this point in the function, we have processed all split definitions and have created a new animation set for each. We will now need to register these animation sets with the controller. But before we do, we first need to remove any existing sets. We no longer need these old animation sets as they were merely the source data containers for the new animation sets we have created.

```
    // Release all animation sets from the animation controller
    // (Note: be careful because the GetNumAnimationSets result,
    // and the values passed in to GetAnimationSet alter ;)
    for ( ; m_pAnimController->GetNumAnimationSets() > 0; )
    {
        // Retrieve the animation set we want to remove
        m_pAnimController->GetAnimationSet( 0, &pAnimSet );
        // Unregister it (this will release inside anim controller)
        m_pAnimController->UnregisterAnimationSet( pAnimSet );
```

```
        // Now release the last known copy
        pAnimSet->Release();

    } // Next Animation Set
```

We now have an empty animation controller and it is time to register our new animation sets. Remember that the Reserved member of each split definition structure was used to store an interface pointer to the animation set that we created for that definition. So all we have to do now is loop through each split definition and register the animation sets stored there with the controller.

```
    // Now add all the new animation sets back into the controller
    for ( i = 0; i < nDefCount; ++i )
    {
        // Retrieve back from the structure
        pDestAnimSet = (LPD3DXKEYFRAMEDANIMATIONSET)pDefList[i].Reserved;
        if ( !pDestAnimSet ) continue;

        // Register with the controller (addrefs internally)
        m_pAnimController->RegisterAnimationSet( pDestAnimSet );

        // Release our local copy and clear the list item just in case
        pDestAnimSet->Release();
        pDefList[i].Reserved = NULL;

    } // Next Definition
```

Now we have an animation controller with new animation sets defined by the split definition array we passed in. Since we have removed all the old animation sets, there will not currently be an animaton set assigned to any track. Therefore, this function will perform a default assignment of the first set to track 0 on the mixer.

```
    // Set the first set in track 0 for default
    m_pAnimController->GetAnimationSet( 0, &pAnimSet );
    m_pAnimController->SetTrackAnimationSet( 0, pAnimSet );
    pAnimSet->Release();

    // Success!!
    return D3D_OK;
}
```

That was a fairly large function and one you will probably not call directly very often. As discussed, Lab Project 10.2 is a splitter tool which is essentially built entirely around this one function call. It provides a nice way around a problem for people who have their animation data combined in the X file and wish to divide it into separate sets. ApplySplitDefinitions is a utility function and should not be used in a time critical portion of your code. Also note that if you intend to save the actor back out to an X file, then you must make sure that the actor is created in managed mode. Finally, if you wish to use this function, you will want to disable the call to ApplyCustomSets in CActor::LoadActorFromX since this function expects to work with the data extracted from the X file. It has no concept of callback keys (as they are never specified in an X file) and it has no concept of our CAnimationSet object either.

Supporting CActor Instancing

Although we have finished adding animation support to the CActor class, we will use this lab project to add some much needed support to our applications for the correct instancing of animated actors. As we know from previous lessons, the ability to have several objects in our scene that have unique positions but share the same mesh data is vitally important. Instancing allows us to avoid storing multiple copies of identical geometry (and other properties) in memory. Indeed, we have used mesh instancing going back all the way to some of the first projects we have done together. This of course is why our CScene class represents objects using the CObject class. The CObject class contains a world matrix that describes the position of the object in the world. It also has a pointer to a separate mesh or actor which points to the actual model space data of the object's associated mesh(s).

In the case of a simple mesh, instancing is easy. If we have a mesh of a street lamp that we wish to place at multiple locations in the scene, we can load the street lamp mesh once and then have several CObject's reference that mesh. Each CObject contains its own world matrix that can be used to position the mesh in the correct place in the scene. In the case of simple static meshes like this, all we have to do before we render an object is set its world matrix on the device and then render the mesh it points to. Even if several CObject's point to the same mesh, as long as we set each object's world matrix prior to rendering its associated mesh, the pipeline will correctly transform and render the mesh in the correct position in the scene. Since this is how we render non-instanced objects as well, our application does not care whether it is the only object referencing that mesh data or not. It all works the same way.

When dealing with actors, instancing support is not quite so automatic or free from CPU overhead. An actor contains a hierarchy of matrices which must all be updated to position the actor in the scene using the associated object's world matrix. Unlike a single mesh object where we can just set the object's matrix as the current world matrix on the device and then render that mesh, when we set the world matrix of an object that stores an actor, we do not simply set the object's matrix as the device world matrix. In fact, we use the CActor::SetWorldMatrix function, passing in the object's world matrix. This function will multiply that world marix with the root frame's relative matrix, changing the root frame of reference for the entire hierarchy. It will then need to perform a full traversal of the hierarchy to rebuild the absolute world matrices stored at each frame. Once done, each frame in the hierarchy will contain its absolute world matrix and thus any mesh attached to that frame will have the matrix it needs to set on the device prior to being rendered.

After the hierarchy matrix update to world space, the rendering step involves nothing more than traversing the hierarchy and searching for mesh containers. When a mesh container is found, the absolute matrix stored in the owner frame will be the world matrix that must be set on the device prior to rendering that mesh. As long as we call the CActor::SetWorldMatrix function and pass in the object's world matrix prior to rendering that actor, we will update the matrices of the actor that describe where the mesh components comprising the actor should be positioned in the scene (using the object's matrix as a frame of reference). If multiple objects reference the same actor, it no longer matters. As long as we update the matrices of the actor using each object's world matrix prior to rendering it.
Of course, we have to be aware of the potential overhead that comes with referencing an actor, even though this same overhead would be incurred if we loaded five unique actors. For example, if we have five CObject's that point to the same actor, then we will have to set the world matrix for the actor five times, once for each time we render the actor for a given object. This means five hierarchy traversals

will be peformed to update the matrices of the actor such that it can be rendered in five different locations. However, even if we used five unique actors, we would still need to traverse each actor's hierarchy every time its associated object world matrix was updated. Of course, this is where the subtle difference comes into play between using five separate actors and referencing the same actor five times.

In the case when five unique actors are being used, each actor would only have to have its hierarchy updated if its associated world matrix had been updated. For example, if only one of the objects had its matrix updated, the other four actors could be left alone since their matrices are currently up to date. Only the actor whose world matrix has been updated would need to have all its matrices rebuilt before being rendered. In the case of one actor being referenced by five different objects, even if none of those objects have had their world matrices changed since the last update, the matrices of the actor will always have to be rebuilt five times, once for each object that uses it. This is because we must use the actor's matrices to take a snapshot of the actor from multiple locations in the scene (once for each object). Thus we will always have to rebuild the matrices of the actor every frame for every object that references it. While this usually means many more hierarchy traversals than in the case of loading the geometry multiple times, the memory savings is significant and often the only practical solution.

> **Note:** It should be noted that a Clone function could be added to our CActor to create a new actor that is a copy. This new actor, while having its own unique hierarchy and matrices, could have mesh containers that point to the original mesh containers in the actor from which it was cloned. The result is the ability to have multiple CActor's that all share the same mesh data (model space vertices and indices) but that have their own unique hierarchies. This is like a hybrid of the non-referenced and referenced approaches. Memory would still be consumed by the multiply hierarchies maintained by each actor, and this might not always be practical, but the mesh data would be shared. With such a system, any actors that have not had their associated object's world matrix updated would not need to have their hierarchy matrices updated needlessly. We will leave the creation of such a function as an exercise for the reader. In our applications, we will be using classic referencing when more than one CObject points to a given actor. As such, there will be only one copy of an actor's hierarchy and mesh data in memory at any given time.

Instancing actors is quite simple if the actor is not animated. For example, the following code snippet shows how the objects could be rendered inside our CScene::Render function. We are assuming that managed meshes are being used to remove the setting of textures and materials from the listing and to simplify the point we are trying to make clear.

```
for ( i = 0; i < m_nObjectCount; ++i )
{
   CObject  * pObject = m_pObject[i];
   if ( !pObject ) continue;

   // Retrieve actor and mesh pointers
   CActor   * pActor = pObject->m_pActor;
   CTriMesh * pMesh  = pObject->m_pMesh;
   if ( !pMesh && !pActor ) continue;

   // Set up transforms
   if ( pMesh )
   {
      // Setup the per-mesh / object details
      m_pD3DDevice->SetTransform( D3DTS_WORLD, &m_pObject[i]->m_mtxWorld );
      m_pD3DDevice->SetFVF( pMesh->GetFVF() );
```

```
      // Draw Mesh (managed mode example)
      pMesh->Draw();

   } // End if mesh

   else

   {
      // Set world matrix and update combined frame matrices.
      pActor->SetWorldMatrix( &pObject->m_mtxWorld, true );

      // Draw Actor (Managed mode example)
      pActor->DrawActor()

   } // End if actor

} End for each object
```

As the above code shows, we loop through each object in the scene and fetch either its mesh or actor pointer. If the object contains a pointer to a mesh, then the object's world matrix is bound to the device and the mesh is rendered. If the object points to an actor, the CActor::SetWorldMatrix function is used. This function does not actually set any matrices on the device, it combines the passed world matrix with the root frame matrix, changing the frame of reference for the entire actor. The 'true' parameter that is passed instructs the function to then traverse the hierarchy and rebuild all the absolute matrices for each frame in the hierarchy. We then call CActor::DrawActor to render the managed mode actor. This function is the function that traverses the hierarchy and sets the matrices on the device prior to rendering each mesh contained there.

The above code works flawlessly whether actors or meshes are being referenced (or not). The code does not care if a CObject's associated mesh or actor is being used by more than one object because the same rendering logic works in both cases. If animation was not a factor, that is all we would have to do to provide correct instancing support for our actors. However, as we have learned in this chapter, an actor can also contain animation data and it is here that our current instancing method would break if we were to take no additional measures. In fact, the code we have used above was implemented in previous lessons so we have had support for static actor instancing for some time now. With animated actors things get a little more complicated and we will have to perform some additional logic and add two new member functions to CActor to make sure that animated actor instancing is properly done.

Who Controls the Controller?

When an actor contains animation data, we update the animations each frame by calling the actor's AdvanceTime method. This method passes the request on to the underlying animation controller which updates the relative matrices to reflect the changes caused by the advance in the timeline. As previously described, we do this inside the CScene::AnimateObjects function, and our current animation update strategy might look like the following:

```
void CScene::AnimateObjects( CTimer & Timer )
{
    ULONG i;
    static CActionHandler Handler;

    // Process each object for coll det
    for ( i = 0; i < m_nObjectCount; ++i )
    {
        CObject * pObject = m_pObject[i];
        if ( !pObject ) continue;

        // Get the actor pointer
        CActor * pActor = pObject->m_pActor;

        if ( !pActor ) continue;

        // Advance time
        pActor->AdvanceTime( Timer.GetTimeElapsed(), false, &Handler );

    } // Next Object
}
```

As you can see, the above code loops through each CObject in the scene and tests to see if that object uses an actor or a mesh. If the object uses a mesh, then there is nothing to do because a single mesh does not contain animation data. If the object uses an actor, then the actor's animation controller is advanced by calling the CActor::AdvanceTime method. If the actor has no animation data then this is a no-op and we can safely call it for all actors.

The problem happens when multiple CObject's are referencing the same actor. Since the actor houses the animation controller, if multiple objects reference the same actor, they are also referencing a shared animation controller. If we had five objects that all referenced the same actor, the above function would not advance the controller time of five actors, it would advance the time of a single actor's controller five times in a given frame update. Therefore, every object would share the same animaton data and the same position in the animation's timeline. Furthermore, that timeline would be playing back at an incorrect speed because we are advancing the timeline multiple times in a given update (once for each object referencing the actor). This will simply not do. Although we can certainly live with the fact (and will actually often desire it) that all objects that reference the actor will essentially contain the same animation data and will also have to have synchronized timelines, we certainly can not settle for those animatons playing back at faster speeds than intended. The situation obviously gets a lot worse if there were hundreds of ojects referencing that actor as its animation timeline would be updated hundreds of times in a given frame update.

We could work around such a problem by using a 'frame time' variable inside the actor so that any calls to AdvanceTime past the first that have identical frame times are ignored. That way, the actor would ignore any AdvanceTime calls except the first for a given frame update. If this technique were implemented, even if 100 objects called AdvanceTime for a single actor in a given frame update, only the first would be processed. But this is still far from ideal; the system we are going to use will provide much more flexibility. Not only will it allow each object to have independent timelines and animation speeds, it can also give each object that references a single actor its own unique animation data.

The root cause of our problem is that every object shares the same animation controller stored inside the actor they are referencing. However, if we were to change this relationship and instead give each object its own copy of the controller (cloning the original controller that was loaded with the actor data) we remove this conflict. We could update our CObject structure so that it now also maintains a pointer to an ID3DXAnimationController like so:

```
class CObject
{
public:
    //-----------------------------------------------------------------------
    // Constructors & Destructors for This Class.
    //-----------------------------------------------------------------------
    CObject( CTriMesh * pMesh  );
    CObject( CActor   * pActor );
    CObject( );
    virtual ~CObject( );

    //-----------------------------------------------------------------------
    // Public Variables for This Class
    //-----------------------------------------------------------------------
    D3DXMATRIX                  m_mtxWorld;             // Objects world matrix
    CTriMesh                   *m_pMesh;                // Mesh we are instancing
    CActor                     *m_pActor;               // Actor we are instancing
    LPD3DXANIMATIONCONTROLLER   m_pAnimController       // Instance Controller
};
```

Since each object has its own controller, it has the ability to move that timeline along at any speed and to whatever position it desires. As mentioned, we could conceivably even generate completely unique animation data for each object as long as the names of the frames comprising the actor's hierarchy are known. For this to work, before we update the animation time of each object, we would first need to attach it to the actor. We can then use the actor's animation functions to alter the timeline of the controller for that particular object. We may also want to allow the attaching of the controller to automatically synchronize the relative matrices of the hierarchy to the data contained in the controller. Simply storing the controller's pointer inside the actor will not update the hierarchy to reflect the current pose described by the timeline of that controller.

Our new version of CScene::AnimateObjects is shown below. It has an additional line of code that calls a CActor method yet to be implemented (AttachController). For now, do not worry about how the object is assigned its own animation controller; we will see this in a moment when examining the changes to the CScene IWF loading code. For now just know that if multiple CObject's reference the same animated actor, the actor will no longer contain its own animation controller -- the CObjects will have their own copy.

```
void CScene::AnimateObjects( CTimer & Timer )
{
    ULONG i;
    static CActionHandler Handler;

    // Process each object for coll det
    for ( i = 0; i < m_nObjectCount; ++i )
    {
```

```
        CObject * pObject = m_pObject[i];
        if ( !pObject ) continue;

        // Get the actor pointer
        CActor * pActor = pObject->m_pActor;
        if ( !pActor ) continue;

        if ( pObject->m_pAnimController )
            pActor->AttachController( pObject->m_pAnimController, false );

        // Advance time
        pActor->AdvanceTime( Timer.GetTimeElapsed(), false, &Handler );

    } // Next Object
}
```

As you can see, we still loop through each object searching for actors to advance their timelines. However, when we find an object that has its own animation controller we attach it to the actor. This function simply copies the passed animaton controller pointer into the actor's controller member variable (as it no longer contains its own animation controller). Once this has been done, we can then use the CActor::AdvanceTime function (and all the other CActor controller manipulation methods) to advance the timeline of the controller. Everytime we attach a new controller to the actor it overwrites the pointer of the controller that was previously stored.

The second parameter to the CActor::AttachController is a Boolean variable that dictates whether we would like the relative matrices of the actor's hierarchy to be updated to reflect the pose of the actor as defined by the current position of the controller's timeline. At this point we do not because we are simply using the actor interface to advance the timelines of each object's controller. You will see in a moment how we will call this function again in the CScene::Render function prior to rendering each CObject. That is, before we render each object in our world, if it is an actor, we will attach the object's controller to the actor, this time passing true as the Boolean parameter. This will cause the relative matrices of the actor to be synchronized to those described by the controller so that the actor is in the correct pose described by the currently attached controller. We will then set the actor's world matrix to initiate a traversal of the hierarchy and a rebuilding of the absolute matrices. These world matrices will be populated as described by the controller. We will then render the actor as we usually would. The snippet of rendering code in our CScene::Render function now looks something like this:

```
for ( i = 0; i < m_nObjectCount; ++i )
{
   CObject  * pObject = m_pObject[i];
   if ( !pObject ) continue;

   // Retrieve actor and mesh pointers
   CActor   * pActor = pObject->m_pActor;
   CTriMesh * pMesh  = pObject->m_pMesh;

   if ( !pMesh && !pActor ) continue;

   // Set up transforms
   if ( pMesh )
   {
```

```
      // Setup the per-mesh / object details
      m_pD3DDevice->SetTransform( D3DTS_WORLD, &m_pObject[i]->m_mtxWorld );
      m_pD3DDevice->SetFVF( pMesh->GetFVF() );

      // Draw Mesh (managed mode example)
      pMesh->Draw();

   } // End if mesh

   else

   {
      // Attach controller with frame synchronization
      if ( pObject->m_pAnimController )
          pActor->AttachController( pObject->m_pAnimController, true )

      // Set world matrix and update combined frame matrices.
      pActor->SetWorldMatrix( &pObject->m_mtxWorld, true );

      // Draw Actor (Managed mode example)
      pActor->DrawActor()

   } // End if actor

} End for each object
```

As you can see, the only line added is the one that attached the controller of an object to the actor prior to rendering it.

Let us now examine the two new functions we will add to CActor that will allow us to attach and detach an animation controller.

CActor::AttachController

AttachController replaces the animation controller pointer currently stored in the actor with the one that is passed in. This is our means of connecting each CObject's controller to the actor so that we can use its methods to manipulate the timeline and update the actor's matrices for the given instance.

The function takes two parameters. The first is a pointer to the new ID3DXAnimationController that will be stored in the actor. Any previous controller stored in the actor will be released. If the controller currently stored in the actor is not being referenced reference elsewhere then it will be freed from memory. Usually, the controller stored here will also have an interface reference in the CObject that is referencing the actor, so in most cases we are simply releasing the actor's claim on that interface. The second parameter is a Boolean variable that controls whether the function should take the additional step of updating the relative matrices of the hierarchy so that they are synchronized to the timeline of the new controller.

As discussed previously, this function is called twice for each CObject that references an actor. The first time it is called (inside CScene::AnimateObjects) we pass false as the second parameter because we do not want to synchronize the matrices of the actor to the controller at this point. We simply want to attach

the controller so that we can use the actor interface to advance the timeline. The second time we call this method for each CObject is when we are about to render the object. This time, we attach the controller and pass true as the Boolean parameter so that the relative matrices are synchronized to the positions described by the animation data in the controller for the current position on the timeline. We then set the world matrix of the actor, which concatenates the relative matrices in their newly animated states, to build the absolute world matrices of the actor at each frame. Finally, we render the actor.

The first section of code releases the actor's claim on any controller interface that is currently being referenced and sets the controller pointer to NULL.

```
void CActor::AttachController( LPD3DXANIMATIONCONTROLLER pController,
                               bool bSyncOutputs /* = true */ )
{
    // Release our current controller.
    if ( m_pAnimController ) m_pAnimController->Release();
    m_pAnimController = NULL;
```

The next piece of code is all in a conditional block that is only executed if the first parameter to the function is not NULL. An application could pass NULL as the first parameter instead of a valid pointer to an animation controller. The result would be the removal of the controller currently being used by the actor and the setting of this pointer to NULL.

Provided the application has passed a valid controller pointer the next code block will be executed. It stores the controller poiner in the actor member variable and increases its reference count. If the second parameter to the function was false, then our job is done. However, if true was passed, then we want the controller we have just attached to update the relative matrices of the hierarchy.

How does this happen? Well, we know that when we advance the time of the animation controller, this will automatically cause the controller to update the relative matrices (animation outputs) of the hierarchy it is animating. However, we do not want to physically advance the time of the controller. That is not a problem. All we have to do is call AdvanceTime and specify that we would like the timeline advanced by zero seconds. The remainder of the function is shown below:

```
    if ( pController )
    {
        // Store the new controller
        m_pAnimController = pController;

        // Add ref, we're storing a pointer to it
        m_pAnimController->AddRef();

        // Synchronise our frame matrices if requested
        if ( bSyncOutputs ) m_pAnimController->AdvanceTime( 0.0f, NULL );

    } // End if controller specified
}
```

By calling AdvanceTime and passing in zero seconds, we do not erroneously alter the position of the timeline (after all, we do that in CScene::AnimateObjects). However, calling this function does instruct the controller to fetch the SRT data for each matrix for the current the position on the timeline. This SRT

data is retrieved from the controller's animation data and used to rebuild the relative frame matrices. We will see in a moment when we examine the loading code that while the CObjects now own their own controllers, these controllers are all clones of the original controller that was created when the actor was first loaded from the X file. Therefore, all the controllers will have correctly had all the frames in the hierarchy that require animation registered as animation outputs.

The next function we will add to the CActor class is called DetachController and is the mirror of the current function. It just sets the controller pointer of the actor to NULL and returns the controller interface currently being used by the actor back to the application. After this function has been called, the actor's internal controller pointer will be set to NULL.

Before we discuss why the application might wish to fetch the controller currently being used by the actor and also have it removed from the actor at the same time, let us look at the function code.

CActor::DetachController

DetachController returns an interface to the controller currently being used by the actor and sets the controller pointer member variable to NULL. As such, the actor has no controller attached after this call. Here is the code:

```
LPD3DXANIMATIONCONTROLLER CActor::DetachController( )
{
    LPD3DXANIMATIONCONTROLLER pController = m_pAnimController;

    // Detach from us
    m_pAnimController = NULL;

    // Just return interface
    return pController;

}
```

The function takes no parameters and returns an ID3DXAnimationController interface pointer back to the caller. Normally, when we remove an object's claim on an interface we would call Release on that interface before setting the pointer to NULL. Notice that we do not do so in this case -- we simply set the pointer to NULL. This is because under normal circumstances, when a function returns an interface pointer, it would increase the reference count of the interface before returning that pointer. In this case, it is unnecessary. Normally we would first increase the reference count when we make the interface pointer copy that will be returned back to the application and would then call Release on that interface before setting the actor's pointer to NULL. Since these two steps cancel each other out, we will simply set the actor pointer to NULL and transfer ownership of the interface reference to the function caller. As the actor loses the reference to that interface, the caller will gain it. The calling function should call Release on this interface when it no longer wishes to use it.

In order to understand why we might need to detach a controller, we need to understand the loading process. That is what we will discuss next.

In our current applications, the only way multiple instances of an actor (or even multiple actors) can be created is if we are loading the scene file from an IWF file using CScene::LoadSceneFromIWF. This is because the alternative method of scene loading we provide (CScene::LoadSceneFromX) assumes that the entire scene is stored in a single X file and thus, only one actor will ever be created. In the case of IWF loading, the situation is different. While the IWF file may contain several internal meshes, these meshes will be loaded and stored as single mesh CObjects in the scene. We can think of these internal meshes as the brushes that you might place in GILES™ since this geometry is stored directly in the IWF file and can contain no animation or hierarchy data. The IWF function that picks these internal meshes out of the IWF file during the loading process is the CScene::ProcessMeshes function. For each mesh found in the IWF file, its data will be copied into a new CTriMesh object and stored in a new CObject. Our actor class is not used for static meshes stored in the IWF file.

However, we do use actors to load reference entities that are stored in the IWF file. A reference entity in an IWF file is an entity that contains a world matrix and a filename. This filename is the name of an X file that contains an object or object hierarchy positioned in your scene using a tool like GILES™. There are many benefits to using references in this way.

For example, imagine you have a very nice X file which contains the representation of a tree. This tree might be comprised of several meshes stored in a hierarchy inside the X file. Furthermore, this X file might also contain animation data that has been hand crafted to allow the tree to sway back and forth. If you were to import this X file into GILES™ as a regular mesh, the seperate meshes comprising the tree would all be collapsed into a single mesh and the hierarchy and animation data would be lost (as GILES™ has no native support for them). This is most likely not the outcome you want.

A better approach would be to design your animated X file in a package such as 3D Studio MAX™ or Maya™ and export the data out to an X file. You can then place multiple references to this X file in the scene using the GILES™ reference entity. For example, you might place 100 of your trees in the level. The geometry of these trees will never be stored in the IWF file. All that will be stored in the IWF file when you save the scene in GILES™ is the position and orientation of each reference (its world matrix) and the name of the X file that should be loaded by the application. This is helpful because it allows you to design your individual objects in your favorite modeling packages and use GILES™ as a simple scene placement tool. You can even create an IWF file in GILES™ which has no internal geometry whatsoever. It might just be a collection of reference entities describing where these external X file objects should be positioned in the scene.

When parsing reference entities during the loading of an IWF file our actor class will be used. The CScene::ProcessReference function is called during the loading process to extract the reference entity information. For each entity loaded we will create a new CObject. There will be a world matrix (describing its position in the scene) for that reference that we will store in the CObject's world matrix. The filename can be passed into a new CActor using CActor::LoadActorFromX as we have done in past lessons. This actor pointer will then be stored in the new CObject and added to the scene's CObject array.

None of this is new to us, but the next part is. When the LoadActorFromX method returns, the actor will have been fully populated with hierarchy data and an animation controller (if animation existed in the X file). In the past, we have simply assumed that an actor will never be referenced by more than one object

and as such, it was fine for the actor to own the animation controller. We now know however, that if more than one CObject is referencing the actor, the actor should no longer own the controller. Instead, each CObject will have its own copy of the original controller. (If an actor is only referenced by one CObject, then we should just let the actor own the controller and we can set the CObject's controller to NULL.)

Essentially, we have created two operating modes for CActor. If only one CObject references it, it will be allowed to own the controller it was originally assigned in the CActor::LoadActorFromX function. However, as soon as we discover that we are about to create an object that references an actor we have already loaded, we will detach the controller from the actor and store it in the CObject. For every CObject created that references that same actor, we will clone this controller and assign it to the CObject's controller pointer.

See this in action will make it much easier to understand, so let us now have a look at the updated code to the CScene::ProcessReference function. This is the function that is called during the loading of an IWF file to extract the information for each reference entity found in the file. This function is called from CScene::ProcessEntities since references are a type of IWF entity.

CScene::ProcessReference

This function is called by the CScene::ProcessEntities function each time a reference entity is encountered. The function is passed a ReferenceEntity structure which contains information about the reference (filename, etc.), and a reference world matrix describing where it should be positioned in the scene.

The first thing we do is test to see that the ReferenceType member of the ReferenceEntity structure is set to 1. If it is not, we return. A setting other than 1 indicates that this is an internal reference instead of an external reference. Internal references are not supported by our applications at this time.

```
bool CScene::ProcessReference( const ReferenceEntity& Reference,
                               const D3DXMATRIX & mtxWorld )
{
    HRESULT                     hRet;
    CActor                    * pReferenceActor      = NULL;
    LPD3DXANIMATIONCONTROLLER   pController          = NULL;
    LPD3DXANIMATIONCONTROLLER   pReferenceController = NULL;

    // Skip if this is anything other than an external reference.
    if (Reference.ReferenceType != 1) return true;
```

We will now extract the filename of the X file we need to load from the ReferenceName member of the input structure. As this will contain just the filename (not the complete path) we will need to append this filename to the data path string currently being used by the application. The CScene::m_strDataPath member is a string that contains the current folder where application data is stored (e.g., '\MyApp\MyData\'). This is the folder where our application expects to find textures and X file resources.

```
    // Build filename string
    TCHAR Buffer[MAX_PATH];
    _tcscpy( Buffer, m_strDataPath );
    _tcscat( Buffer, Reference.ReferenceName );
```

At this point we have the complete filename of the X file we wish to load (including path) stored in the local Buffer variable.

We now loop through every actor that is currently in the scene's CActor array and test to see if an actor already exists with the same name. If so, then it means that we have already loaded this X file for a previous actor and do not wish to load it again.

```
    // Search to see if this X file has already been loaded
    for ( ULONG i = 0; i < m_nActorCount; ++i )
    {
        if (!m_pActor[i]) continue;
        if ( _tcsicmp( Buffer, m_pActor[i]->GetActorName() ) == 0 ) break;

    } // Next Actor
```

At this point, if loop variable 'i' does not contain the same value as m_ActorCount then it means the above loop exited prematurely in response to finding an existing actor created from the same X file. When this is the case, we do not want to load the X file again; instead we will create a new CObject that references the current actor. However, there are some things we now wish to consider.

We will first need to find the CObject that currently contains a pointer to this actor and test to see if its animation controller pointer is NULL. If it is, then it means that the object is the only one that currently references the actor and at present, the actor owns its own controller. When this is the case, we will detach the controller from the actor and store it in that object instead. We will then clone this controller and store it in the object reference we are about to create. This is essentially the code that recognizes when more than one CObject is referencing an actor and removes the controller management responsibilities from the actor in favor of the object.

Next we see the code that executes this step. First we get a pointer to the actor that our new object is going to reference and then get a pointer to the pre-existing CObject that currently owns it.

```
    // If we didn't reach then end, this Actor already exists
    if ( i != m_nActorCount )
    {
        // Store reference Actor.
        pReferenceActor = m_pActor[i];

        // Find any previous object which own's this actor
        for ( i = 0; i < m_nObjectCount; ++i )
        {
            if (!m_pObject[i]) continue;
            if ( m_pObject[i]->m_pActor == pReferenceActor ) break;

        } // Next Object
```

Notice that the above code exits as soon as any CObject that currently references the actor is found. If multiple CObjects already exist at this point, and they all reference the same actor, then it does not matter which one we use. We are only going to use it to clone its controller into the new CObject reference we are about to create. If however, the object that we find has its animation controller pointer set to NULL then we know that is currently the only CObject that exists that currently references the actor. As such, the actor currently manages the controller. In this case we will detach the controller and assign it to this object. We will then clone this controller and assign the clone to the new object reference we are about to create.

The next code block is executed if an object was found in the above loop. (This should always be the case since there is currently no way that a CActor can be created that is not also assigned to a CObject.) First we get a pointer to that object and test to see if its animation controller pointer is set to NULL. If so then we know it is currently the only object that references this actor and therefore the actor currently contains the controller. As we are just about to create a new object reference to this actor we know the actor mode needs to change. Therefore, we will detach the controller from the actor and store it in the pre-existing object that we found.

```
        // Did we find an object?
        if ( i != m_nObjectCount )
        {
            CObject * pReferenceObject = m_pObject[i];

            // Is this the first reference?
            if ( !pReferenceObject->m_pAnimController )
            {
                // If this is the first time we've referenced this actor then
                // we need to detach it's controller and store in the reference
                // object
                pReferenceObject->m_pAnimController =
                                            pReferenceActor->DetachController();

            } // End if first reference
```

We have just switched the previous object from non-reference mode to reference mode by assigning it its own controller. At this point, the actor no longer owns a controller.

Our next step is to create the new object reference we actually entered this function to build. We know that the reference we are processing at this point references an actor that we already have in memory, so we just have to create a new CObject and add a pointer to the actor. We also know that this is not the only object that references the actor, so this new object will need to have its own animation controller as well.

With the previous object reference taken care of, we will prepare the data for the new object reference we are about to create. Our new object will need its own animation controller, so we will clone the animation controller that we just detached from the actor and assigned to the previous object. As the clone function requires that we pass in all the limitations of the new controller at cloning time, we will fetch these limits from the controller that we are about to clone prior to calling the clone method.

```
            ULONG nMaxOutputs, nMaxTracks, nMaxSets, nMaxEvents;
```

```
            // Retrieve all the data we need for cloning.
            pController = pReferenceObject->m_pAnimController;
            nMaxOutputs = pController->GetMaxNumAnimationOutputs();
            nMaxTracks  = pController->GetMaxNumTracks();
            nMaxSets    = pController->GetMaxNumAnimationSets();
            nMaxEvents  = pController->GetMaxNumEvents();

            // Clone the animation controller into this new reference
            pController->CloneAnimationController(    nMaxOutputs,
                                                      nMaxSets,
                                                      nMaxTracks,
                                                      nMaxEvents,
                                                      &pReferenceController );

        } // End if we found an original object reference.

    } // End if Actor already exists
```

We have now seen the code block that is executed if an actor already exists that shares the same X file as the reference entity we are currently processing. If this code has been executed, the local pReferenceController variable stores a pointer to the interface of a cloned animation controller that will be assigned to the new CObject we are about to create. The local variable pReferenceActor also points to the existing actor that we intend to assign to the new CObject.

The next section of code is executed if the X file whose filename is stored in the reference has not yet been loaded into an actor. When this is the case, we create a new actor just like we always have and use the CActor::LoadActorFromX function to load the X file into the actor. We also register the CScene attribute callback function since our application always uses actors in managed mode.

```
    else
    {
        // Allocate a new Actor for this reference
        CActor * pNewActor = new CActor;
        if (!pNewActor) return false;

        // Load in the externally referenced X File
        pNewActor->RegisterCallback( CActor::CALLBACK_ATTRIBUTEID,
                                     CollectAttributeID,
                                     this );

        hRet = pNewActor->LoadActorFromX( Buffer, D3DXMESH_MANAGED,
                                          m_pD3DDevice );
        if ( FAILED(hRet) ) { delete pNewActor; return false; }

        // Store this new Actor
        if ( AddActor( ) < 0 ) { delete pNewActor; return false; }
        m_pActor[ m_nActorCount - 1 ] = pNewActor;

        // Store as object reference Actor
        pReferenceActor = pNewActor;

    } // End if Actor doesn't exist.
```

At this point a new actor has been added to the scene's CActor array and the actor will have been fully populated with the data from the X file. If the X file contained animation data, the actor will currently store a pointer to a controller. That is where it will remain until a second CObject is created that also references it (as shown in the previous code). Finally, at the bottom of this code block we assign the local pReferenceActor pointer to point to this actor. By doing this we know that whichever of the two code blocks were executed above, the pReferenceActor variable will point to the actor that needs to be assigned to the new CObject we are about to create.

All that is left to do is allocate a new CObject structure and add it to the scene's CObject array before assigning the actor and controller pointers to that object. We also store the world matrix of the reference entity that was passed into the function.

```
    // Now build an object for this Actor (standard identity)
    CObject * pNewObject = new CObject( pReferenceActor );
    if ( !pNewObject ) return false;

    // Copy over the specified matrix and store the colldet object set index
    pNewObject->m_mtxWorld        = mtxWorld;

    // Store the reference animation controller and action status (if any)
    pNewObject->m_pAnimController = pReferenceController;


    // Store this object
    if ( AddObject() < 0 ) { delete pNewObject; return false; }
    m_pObject[ m_nObjectCount - 1 ] = pNewObject;

    // Success!!
    return true;
}
```

One important point to make about the above code is that if the second code block is executed (i.e., the actor did not exist prior to the function being called) then the pReferenceController pointer will be set to NULL and we will be assigning NULL to the object's controller pointer also. This is because, at this point only one CObject references that actor and the actor currently manages the only animation controller. This relationship will change if the function is called again later to process another reference to the same X file. In that case, the first code block is executed and the controller is removed from the actor and stored in the object that was created in the prior call. The controller is then cloned and copied into the new CObject that is created at the tail of the function.

For completeness we will also show the CScene::LoadSceneFromX function. It has been updated with an additional line that registers a callback key handler with the actor that is created.

CScene::LoadSceneFromX

Students already familiar with our application framework will know that the CScene::LoadSceneFromX method is called by CGameApp::BuildObjects when the application first starts. It is passed the filename of the X file to be loaded. It is this loading function that is used by Lab Project 10.1 to load a single X

file that contains our small space scene. We will quickly look at this function code since it demonstrates an important topic that was discussed in this workbook. It calls the actor's RegisterCallback function to register a callback key registration function with the actor. When this function is used to load a scene (instead of LoadSceneFromIWF) the scene will only contain a single actor. As only one actor exists, the actor will always own the controller and the CObject that references it will always have its controller pointer set to NULL. This loading function is much simpler since it does not have to perform the controller attach/detach process.

This function is virtually identical to the implementation we looked at in the last workbook. It creates a new CActor and uses the CActor::LoadActorFromX method to load the requested X file. It then adds this actor's pointer to the scene's CActor array. It also allocates a new CObject and attaches the actor to it before adding this new object to the scene's CObject array. Remember, our scene class deals with CObjects at the top level so that it can handle the storage of both mesh hierarchies and single mesh objects in the scene. The scene's CActor array is not rendered from directly; it is where we store pointers to all the actors so we can release them in the scene's destructor.

Finally, this function sets up a few default lights so that we can see what we are rendering. Although the entire function is shown below as a reminder, notice that adding support for animated CActors has required only one new line of code (and even this line is optional). This new code is highlighted in bold in the following listing. It shows how we register the scene's callback key callback function with the actor. The actor will add this function's pointer, along with the passed context data pointer (the *this* pointer), to the actor's CALLBACK_FUNC array. As we saw earlier, when the actor is loading the X file, it will call this callback function and allow the function to add callback keys to any of its registered animation sets. As with the registration of attribute and texture callback functions discussed in the previous lesson, the callback key callback function must be registered before the call to CActor::LoadActorFromX is made.

```
bool CScene::LoadSceneFromX( TCHAR * strFileName )
{
    HRESULT hRet;

    // Validate Parameters
    if (!strFileName) return false;

    // Retrieve the data path
    if ( m_strDataPath ) free( m_strDataPath );
    m_strDataPath = _tcsdup( strFileName );

    // Strip off the filename
    TCHAR * LastSlash = _tcsrchr( m_strDataPath, _T('\\') );
    if (!LastSlash) LastSlash = _tcsrchr( m_strDataPath, _T('/') );
    if (LastSlash) LastSlash[1] = _T('\0'); else m_strDataPath[0] = _T('\0');

    CActor * pNewActor = new CActor;
    if (!pNewActor) return false;

    // Load in the specified X file
    pNewActor->RegisterCallback(CActor::CALLBACK_ATTRIBUTEID,CollectAttributeID,this);
    pNewActor->RegisterCallback(CActor::CALLBACK_CALLBACKKEYS,CollectCallbacks,this );
    pNewActor->LoadActorFromX( strFileName, D3DXMESH_MANAGED, m_pD3DDevice );
```

```
    // Store this new actor
    if ( AddActor( ) < 0 ) { delete pNewActor; return false; }
    m_pActor[ m_nActorCount - 1 ] = pNewActor;

    // Now build an object for this mesh (standard identity)
    CObject * pNewObject = new CObject( pNewActor );
    if ( !pNewObject ) return false;

    // Store this object
    if ( AddObject() < 0 ) { delete pNewObject; return false; }
    m_pObject[ m_nObjectCount - 1 ] = pNewObject;

    // Set up an arbitrary set of directional lights
    ZeroMemory( m_pLightList, 4 * sizeof(D3DLIGHT9));
    m_pLightList[0].Type = D3DLIGHT_DIRECTIONAL;
    m_pLightList[0].Diffuse = D3DXCOLOR( 1.0f, 0.92f, 0.82f, 0.0f );
    m_pLightList[0].Specular = D3DXCOLOR( 1.0f, 0.92f, 0.82f, 0.0f );
    m_pLightList[0].Direction = D3DXVECTOR3( 0.819f, -0.573f, 0.0f );

    m_pLightList[1].Type = D3DLIGHT_DIRECTIONAL;
    m_pLightList[1].Diffuse = D3DXCOLOR( 0.4f, 0.4f, 0.4f, 0.0f );
    m_pLightList[1].Specular = D3DXCOLOR( 0.6f, 0.6f, 0.6f, 0.0f );
    m_pLightList[1].Direction = D3DXVECTOR3( -0.819f, -0.573f, -0.0f );

    m_pLightList[2].Type = D3DLIGHT_DIRECTIONAL;
    m_pLightList[2].Diffuse = D3DXCOLOR( 0.8f, 0.8f, 0.8f, 0.0f );
    m_pLightList[2].Specular = D3DXCOLOR( 0.0f, 0.0f, 0.0f, 0.0f );
    m_pLightList[2].Direction = D3DXVECTOR3( 0.0f, 0.707107f, -0.707107f );

    m_pLightList[3].Type = D3DLIGHT_DIRECTIONAL;
    m_pLightList[3].Diffuse = D3DXCOLOR( 0.6f, 0.6f, 0.6f, 0.0f );
    m_pLightList[3].Specular = D3DXCOLOR( 0.0f, 0.0f, 0.0f, 0.0f );
    m_pLightList[3].Direction = D3DXVECTOR3( 0.0f, 0.707107f, 0.707107f );

    // We're now using 4 lights
    m_nLightCount = 4;

    // Success!
    return true;
}
```

CActor - Conclusion

We have now covered all of the new code that has been added to the CActor class since Lab Project 9.1. While we have certainly added quite a large amount of code, much of it took the form of simple wrapper functions. Our actor now has built-in support for loading multiple mesh hierarchies, and is fully capable of animating of those hierarchies. Despite the myriad of code we studied here in this workbook, the API for the CActor class makes it extremely easy to use. Indeed, it can be initialized with only a few function calls by the application. This makes the CActor class an excellent tool for us as we move forward with our studies and can prove to be a powerful re-usable component in your own game projects. (And it will become even more potent when we add skinning support in the next chapter!)

Lab Project 10.2: The Animation Splitter

Our coverage of Lab Project 10.2 will be different from that of previous lab projects. We will not be covering the code in depth, but will instead discuss implementation at a high level. The reason we have decided to take this approach is that the application is essentially nothing more than a dialog box that calls the CActor functions we have already covered in this workbook. Therefore, you should have no trouble understanding the code should you decide to examine the source projects.

Design

A screenshot from the Animation Splitter is shown in Figure 10.3. The application has two main components. The first is the main render window which all of our applications have used. Just like our previous applications, this window has its contents redrawn each iteration of the game loop in the CGameApp::FrameAdvance method. Unlike our other lab projects however, there is no CScene class or CTerrain class for managing the objects in the world. This is primarily because we only load one X file at a time. The CGameApp object will load a single actor from an X file (inside CGameApp::BuildObjects) and will call CActor::DrawActor in the CGameApp::FrameAdvance method during each iteration of the main game loop. That is about the extent of the scene management that has to be done in this application.

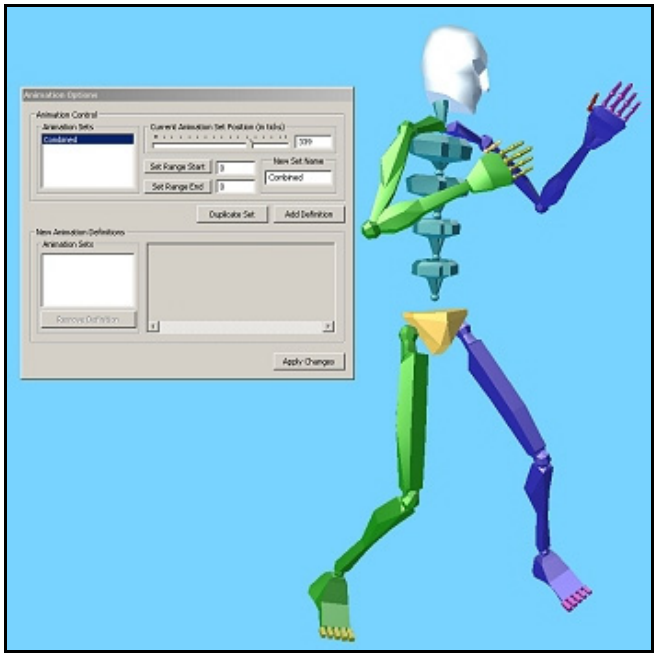

Figure 10.3: The Animation Splitter

In Figure 10.3, we see the main render window with our actor being rendered. In this example, we have loaded a file called ‘Boney.x’ which contains the skeleton of a humanoid character.

The main part of this application that is quite different from previous lab projects is the modeless dialog box that we create to provide a number of controls. We need the dialog because the purpose of this application is to provide a means for editing the animation data stored in the actor. More specifically, the purpose of this tool is to provide a means for us to take an actor's animation set and split it into multiple animation sets. Therefore, we have controls that allow us to navigate to a certain position on an animaton set's timeline and set start and end markers. We can then give this range a name and add it as a new split definition. So the controls on the dialog will not only display the contents of the animation data in the loaded X file, but they will also allow you to alter that data before re-saving the modified X file. Once we have added all the new definitions we wish to add, we can hit the Apply button. At this point, the new animation sets will be created from each split definition and the old animation sets will be released from the actor's controller. It should be obvious that this entire splitting process can be done by the application with a single function call to CActor::ApplySplitDefinitions. In order to understand how the splitter works internally, we shall first examine how it is used. Therefore, this section will also serve as a user handbook for the Animation Splitter.

Using the Animation Splitter

To demonstrate using the animation splitter, we will assume you are working with the X file 'Boney.x'. This X file contains a single animation set (called 'Combined') which contains a number of animation sequences combined into a single set. This is a typical scenario when exporting X files from applications that support only a single animation set and is the reason that this tool was created. It should be noted however, that the application works perfectly fine even if the X file already contains multiple sets. These multiple sets can also be subdivided into more sets if desired.

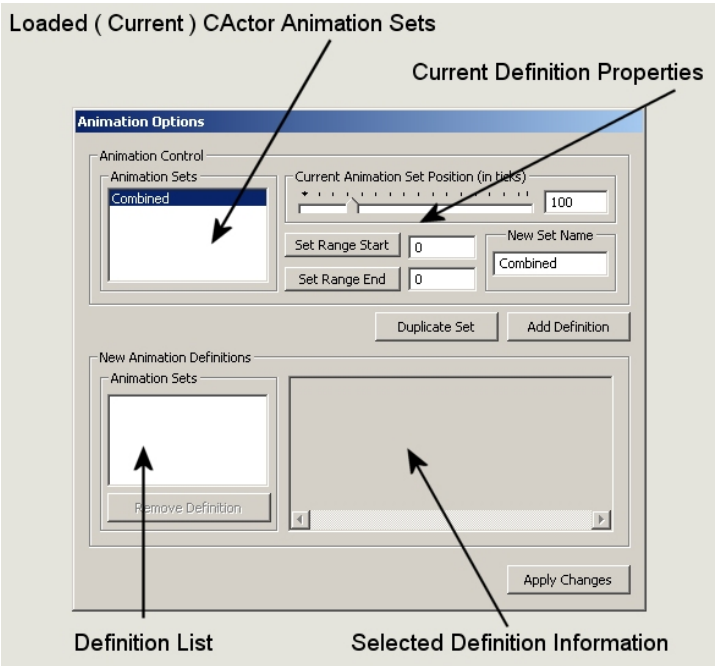

Figure 10.4: The Interface Components

In Figure 10.4 we see how the dialog controls might look after the 'Boney.x' file has been loaded. The uppermost list box contains animation sets that were loaded from the X file and are currently registered with the actor. As you can see, in our example the X file contained only one animation set called 'Combined'. In Figure 10.4 we have selected this animation set, which enables the Split Definition control panel to the right of this list box (the slider, set range buttons, etc.).

Because we have selected the 'Combined' animation set in the list box, this tells the application that the next split definition we add will contain a subset of the animation data in this animation set. Remembering our discussion of CActor::ApplySplitDefinitions, this is the source animation set for the next split definition we are about to construct.

This application allows you to add split definitions to a list one at a time. The bottommost list box on the dialog shows the split definitions you have already added. Obviously, in Figure 10.4 this box is empty as we have not yet added any.

When a source animation set is selected from the top list box, the definition controls become active. We can move the slider to navigate to positions on the animation set's timeline. The 3D render window will reflect the changes in the actor as we drag the slider through the actor's timeline. This makes it easy to visually navigate to the correct position in the animation. The first thing we will want to do is set the start and end markers for this definition. In this example, we will extract an imaginary walking sequence that is positioned between ticks 100 and 200 on the timeline.

First we would move the slider to the start position and press the Set Range Start button. This will record the current position of the slider as the start marker for the definition we are currently building. We would then move the slider position again to where the walking sequence ends (200 in this example) and press the Set Range End button to record this position as the end marker. Figure 10.5 shows the dialog after we have recorded the start and end markers. Notice that the subsection of the timeline is also highlighted in blue on the slider.

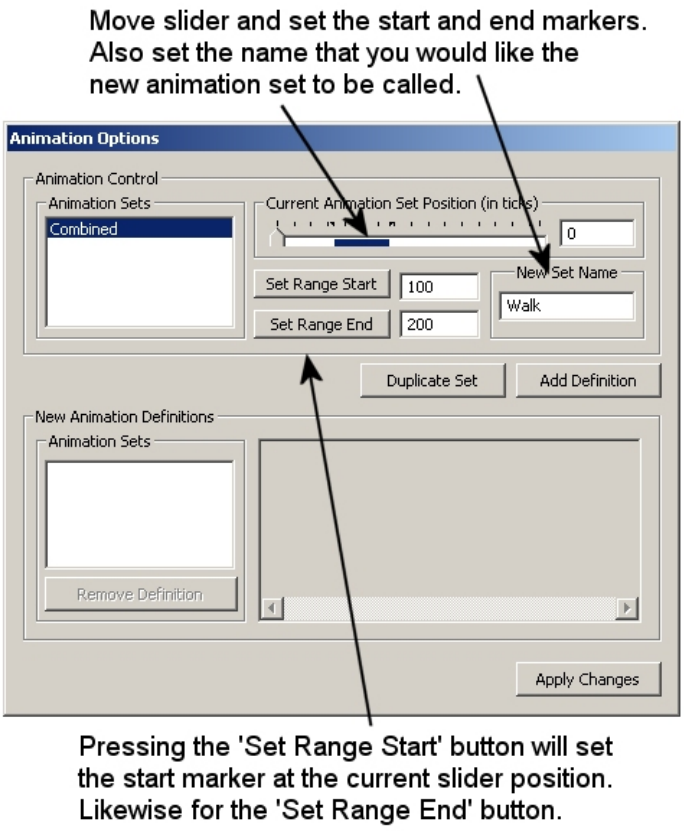

Figure 10.5: Setting up a new Split Definition

We have now marked the exact section of the original animation set that we would like to extract into a new one. That is, we are stating that we will want a new animation set to be built that will only contain the animation data that fell between 100 and 200 ticks on the timeline of the original animation set. We will also want to give this new animation set a name. As you can see in Figure 10.5, we assign it the name ‘Walk’. We just type the name into the text edit box to the right of the Set Range Start and Set Range End buttons.

At this point we have correctly set up the properties for our first split definition. We can now inform the dialog that we wish to add this split definition to a list and move on to defining another one. We add the properties to the split definitions list by pressing the Add Definition button.

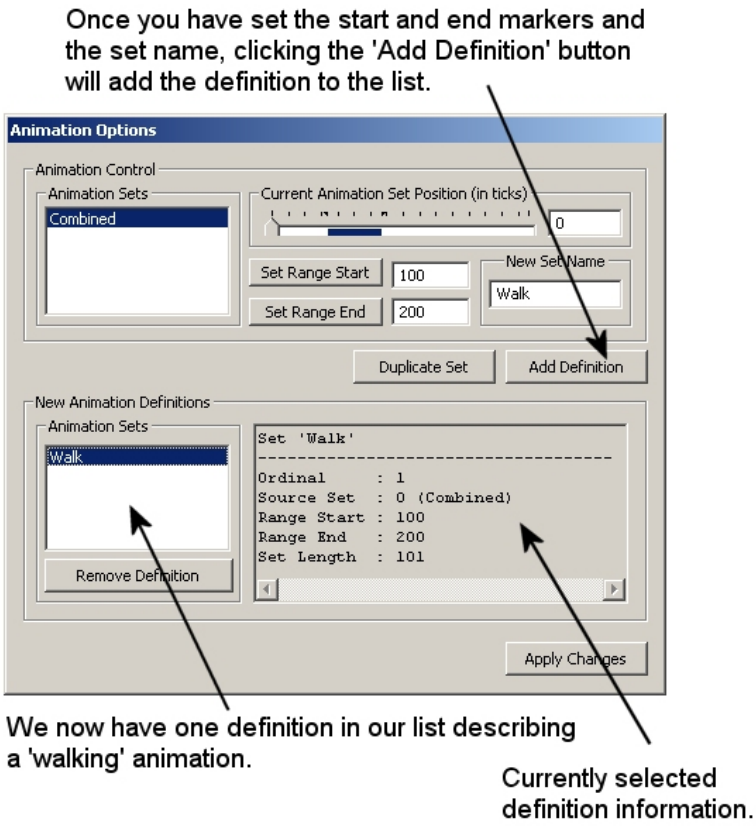

Figure 10.6: Adding the Split Definition

Once the new definition has been added you will see it appear in the bottom list box. As Figure 10.6 shows, this box contains the name of our newly added ‘Walk’ animation. What is important to realize is that at this point, no animation sets have been physically created. All we are doing is adding split definition structures to an array one at a time. This allows the original source animation sets to remain in the controller so that they can be sources for subsequent definitions we may wish to create.

At this point we have added one split definition to our list. Continuing this example we will now definine a second animation set which we will call ‘Run’. For the sake of this explanation, we will imagine that the keyframes for this sequence are also contained in the original ‘Combined’ animation set between tick positions 300 and 400.

With our new ‘Walk’ definition added to the list, we can now set the definition controls for the next definition we wish to add. In Figure 10.7 you can see that the slider has been used to set two new start and end marker positions on the animation timeline. This time, the markers bound a 100 tick range between positions 300 and 400. We then type the name (‘Run’) we would like this animation set to be called in the New Set Name edit box. We have now filled out all the information for our second split definition.

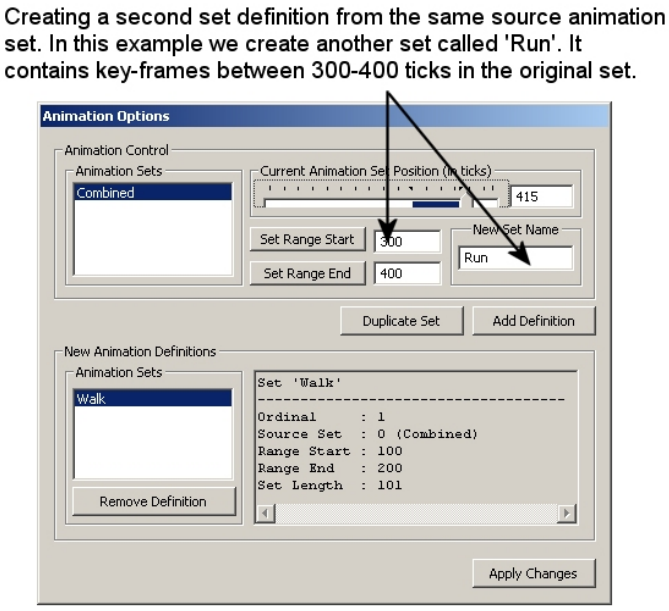

Figure 10.7: Setting up a second Split Definition

With the definition controls now containing the information for our second definition, we add it to the split definition list. We once again press the Add Definition button which creates a new split definition and adds it to the current array. If we look at the bottom list box in Figure 10.8, we can see that after this step is performed there are now two definitions in our list: ‘Walk’ and ‘Run’.

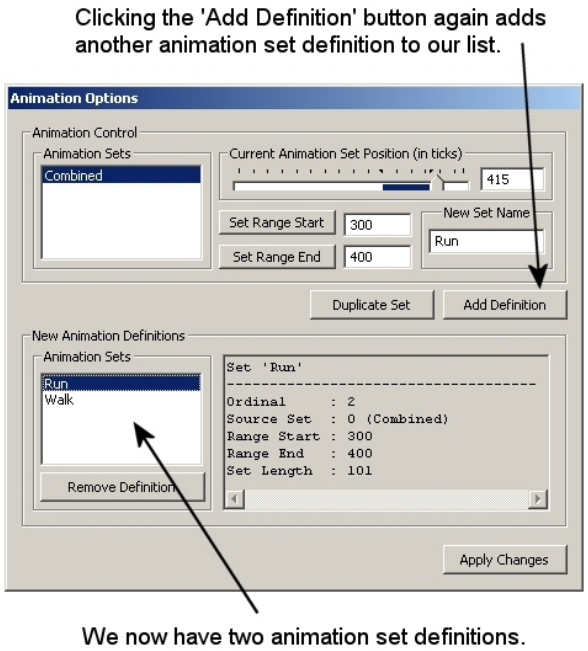

Figure 10.8: Adding the Second Definition

Assuming that we only wish to break the source animation set into two sets in this example, our work is done. It is now time to instruct the application (more specifically, the actor) to apply these split definitions.

When the Apply Changes button is pressed, the current array of split definitions stored in the dialog object is passed to the CActor::ApplySplitDefinitions method. We know that this method will create a new animation set from each definition and register them with the actor's animation controller. At this point, the original animation set(s) will cease to exist. The actor has now been modified in memory and its original animation set(s) have been replaced by the the new animation sets described by the definitions.

Finally, Figure 10.9 shows how the dialog would look after this step. At this point, the definitions list is empty since there are no longer any pending split definitions to process. Those definitions have now been turned into real animation sets and therefore, they are now listed in the top list box instead of the original source set. At this point, you could perform more subdivision steps on these new animation sets, although this is something you will probably not need to do very often.

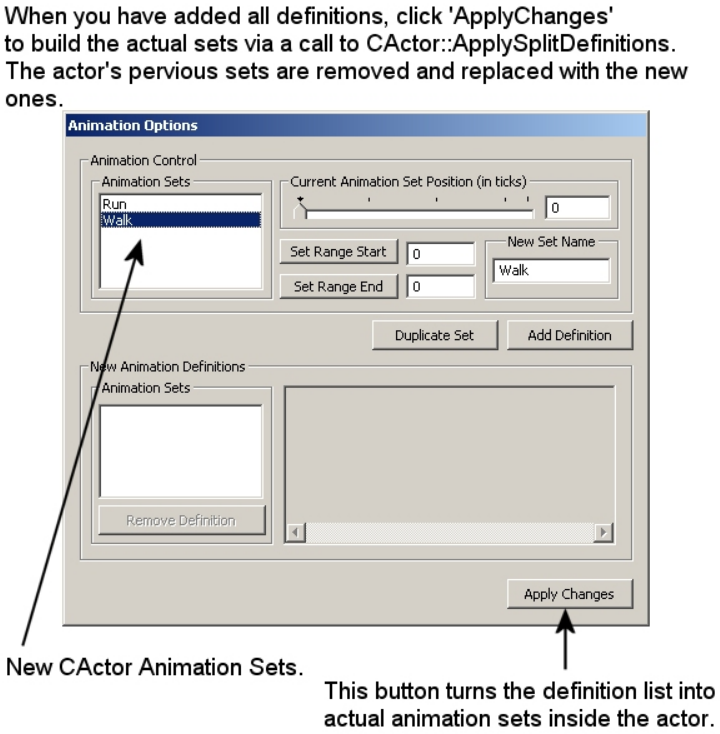

Figure 10.9: Building Animation Sets from Split Definitions

The actor's animation sets have now been changed, but no editing of the original X file has been done. Usually, when splitting animations in this way you will want to save the modified actor back out to the X file. You can do this by selecting the Save option from the file menu in the main render window.

Implementation

Most of the code to this lab project has already been covered, due to the fact that the majority of its functionaility is within CActor. We are also quite familiar with the CGameApp class, having used it in all of our lab projects from the beginning of this course series. Although we are going to make one or two changes to this class, for the most part, it is the same in this project as all the other versions that have come before it. One of the important new changes in CGameApp happens in the CGameApp::CreateDisplay method. This method has been used right from Lesson One to build our enumeration object and create the main render window of the application. Now, at the bottom of that function we also make the following call:

```
m_AnimDialog.InitDialog( m_hWnd, &m_Actor )
```

m_AnimDialog is a variable of type CAnimDialog, which is a class we will write in this lab project. This dialog class is responsible for tracking changes to controls and reflecting those changes back to the actor. The CAnimDialog::InitDialog method is the method that actually instructs the CAnimDialog object to display the modeless dialog on the screen. The dialog box layout is defined as a resource, so controls are automatically created for us when the call to the Win32 function CreateDialogBoxParam is made.

The InitDialog method accepts two parameters that our dialog class will need. The first parameter passed in by CGameApp is the handle of the main render window. Although the dialog will be modeless and not confined to being positioned within the bounds of the render window, the render window will still be its parent and the window that contains the application menu. When the parent window is destroyed, so is the child dialog.

The second paramater passed by CGameApp is the address of the actor that has been loaded. The dialog class will need access to the actor so that it can call its ApplySplitDefinitions function when the Apply button is pressed. It will also need to alter the position of the currently selected animation set in response to the user dragging the slider bar. This will ensure that when the slider bar is moved to a new position, the actor's hierarchy is updated to reflect that position change. Since the CGameApp object is consistently rendering the actor in a loop (via CGameApp::FrameAdvance), those changes made to the actor's hierarchy will be immediately reflected in the render window when the actor is redrawn in the next iteration of the render loop.

> **Note**: We are creating a modeless dialog box so that the program will not have to wait for the user to finish applying definitions before our CGameApp code can continue execution. As soon as InitDialog has created the dialog box, program flow will return back to CGameApp and the main render loop will be entered. The dialox box will remain on the screen processing input and relaying these property changes to the actor.

CAnimDialog – A Brief Overview

We will not step through the code to the CAnimDialog class line by line as we have done in other lab projects. It uses the most basic Win32 concepts (dialogs and controls) and most of the code is just concerned with setting and getting the properties of these controls and handling any messages they send and receive. Win32 controls and dialog box programming is discussed in detail in **C++ Programming for Game Developers: Module II** here at the Game Institute, so we will assume that you have either taken this course already, or that you are familiar with Win32 user interface programming techniques. We will not explain the concepts of Win32 dialog box and control programming here since it falls outside the scope of what we intend to teach in this course. If you are not comfortable with this type of Win32 programming, then we strongly recommend you take the abovementioned course before continuing (or in conjunction with) with your Graphics Programming studies.

Below, we see the CAnimDialog class declaration from CAnimDialog.h. The methods of the class are not shown here, but there are quite a few. Most of these methods are functions that react to control changes while others are called to set the default values for the controls. What we do wish to focus on here are the member variables of this class and how they are used by this object.

```
class CAnimDialog
{

 // Methods not shown here

 private:
    //-----------------------------------------------------------------------
    // Private Variables for This Class
    //-----------------------------------------------------------------------
    HWND                        m_hDialog;
    CActor                    * m_pActor;
    LPD3DXKEYFRAMEDANIMATIONSET m_pCurrentAnimSet;
    ULONG                       m_nCurrentAnimSet;
    ULONG                       m_nStartTick;
    ULONG                       m_nEndTick;

    ULONG                       m_nDefCount;
    AnimSplitDefinition         m_pDefList[MAX_DEFINITIONS];

    HFONT                       m_hFontFW;

};
```

HWND m_hDialog

When the InitDialog method instructs Windows to display the dialog, we are returned a handle to the dialog window. This handle will be needed to send and receive messages to the dialog and its controls. For example, when we wish to send a message to the dialog window (such as to enable the window on screen), we will need to send a message using the SendMessage function. This Win32 function expects the first parameter to be the handle of the window it is expected to send the message to.

CActor * m_pActor
This member is used to cache a pointer to the actor that the dialog controls will manipulate. This pointer is passed in by the CGameApp object when it issues the call to CAnimDialog::InitDialog. This provides the dialog object with a pointer to the same actor the CGameApp object is rendering. The dialog will need access to the actor on several occasions. For example, when the user alters the slider position, the dialog object will need to retrieve the new slider position and use it to set the position of the actor's currently active animation track. The dialog object will also need to communicate with the actor when the user selects a new source animation set. When a new source animation set is selected, the dialog object will retrieve the index of the animation set from the top list box and tell the actor to assign this animation set to track 0. Since we can only be editing using one source animation set at a time, the dialog can always make sure that that the currently selected animation set is bound to track 0 on the actor's mixer. That way, it can safely assume that when the slider position is changed by the user, it has to apply this position change to track 0.

LPD3DXKEYFRAMEDANIMATIONSET m_pCurrentAnimSet
This member is used to store a pointer to the interface of the currently selected source animation set. When the user selects a source animation set from the top list box, it becomes the current animation set. Changing the position slider will allow us to step through the animations in this set. Therefore, when a new set is selected, the dialog object will fetch the animation set interface from the actor and store it here. Now it can assign it to track 0 on the mixer and extract other information from it as needed to populate the controls. For example, the slider bar must represent the timeline of the animation set, so it will be necessary to extract the period of the animation set, convert it to ticks and store this range in the slider control.

ULONG m_nCurrentAnimSet
This member is used alongside the previous member. Where as the previous member stores the interface pointer of the currently selected source animation set, this member stores the index of this animation set assigned to it by the animation controller. This allows us to reference this animation set in the actor by index.

ULONG m_nStartTick
This member will be used to store the current value for the Set Range Start edit box. This is the currently set start marker position used for marking a range on the current animation set's timeline. This value will be used as the start marker when the definition is added to the list.

ULONG m_nEndTick
This member will be used to store the current value for the Set Range End edit box. This is the currently set end marker position used for marking a range on the current animation set's timeline. This value will be used as the end marker when the definition is added to the list.

ULONG m_nDefCount
This member contains the number of split definitions that have been added to the dialog's split definitions array. This will be zero when the application starts.

AnimSplitDefinition m_pDefList[MAX_DEFINITIONS]
This is an array of AnimSplitDefinition structures. Each element in this array will have been added in response to the Add Definition button being pressed on the dialog. When that happens, the split properties (end marker position, start marker position, and set name) are copied from the dialog controls into a new AnimSplitDefinition structure in this array. The definition count is then incremented.

HFONT m_hFontFW;
This is a handle to the font we use in the dialog.

While we will not look at all the helper functions for this class, we will look at a few. The first function we will look at is the dialog box procedure. This is the method that is sent the messages that are generated by the controls (sent via the operating system).

CAnimDialog::OptionsDlgProc

This function handles the basic message processing for the dialog. It is called by Windows whenever a message is generated for our dialog box. For example, if the user pressed the Apply button, a WM_COMMAND message will be sent to this function. The message will also be accompanied by the ID of the control that sent it. For example, if a WM_COMMAND message has been triggered by a control with the id IDC_APPLY, we know the user has pressed our Apply button because IDC_APPLY is the id we assigned to that button in the resource editor.

Although this function is simple (it is just a switch statement that calls processing functions) we will discuss the messages it recieves and the helper functions it calls as a result. We will not be covering those helper functions in this workbook since their code is very straightforward. You should have no problem looking through the source code since it is just basic Win32 controls programming. This function will show you how and why those helper functions get called. We will also explain here what those helper functions actually do. This will give you a better overview of the entire application without us having to examine code.

The CAnimDialog::OptionDlgProc function is called by the operating system (indirectly via a static routing function) whenever a message is generated for the dialog window or one of its controls. We can simply process the messages we want and return false for any that we do not. We return false to indicate that we did not handle the message and that Windows should still perform default processing on it. Luckily, the OS implements a lot of default behavior so that the dialog window knows when to paint itself (for example) or when it is overlapped by other windows. Therefore, we only have to intercept messages which we intend to process using our own additional logic.

When this function is called it is passed the handle of the window that generated the message, the message itself (Windows defines a large set of message codes as UINTs) and two DWORD parameters (wParam and lParam). The contents of the final two parameters will depend on the message being received, as we shall see.

The first section of the code sets up a switch statement to test the Message parameter for messages we are interested in.

```
BOOL CAnimDialog::OptionsDlgProc( HWND hWnd, UINT Message,
                                  WPARAM wParam, LPARAM lParam)
{
    ULONG  i;

    switch ( Message )
    {
    case WM_INITDIALOG:

        // Store dialog here since this is fired before the
        // CreateDialog has returned
        m_hDialog = hWnd;

        // Populate the dialog
        PopulateOptions();

        // Set focus for us.
        return TRUE;
```

The first case happens when the dialog window has received the WM_INITDIALOG message. The operating system will send this message to all dialog box windows just after the window has been created but before it has been displayed. This allows you to perform some default initialization such as filling up the list box controls with the information you want in them. As you can see, we do exactly that.

First we store the handle of the dialog window in the CAnimDialog's member variable, so this C++ object always has a handle to the GUI element to which it is attached (i.e., the dialog window itself). This is the first time we get to know about the handle of the window because the operating system has just created it for us. We store it because we will need it later when sending messages to our controls. Remember, at this point, Windows has created our dialog box and all the control windows that were specified in the resource template. However, these controls and the dialog window are not visible to the user at the moment. All controls (list boxes for example) will also be empty of content at this point.

After we have stored the window handle, we call the PopulateOptions method. This method does quite a bit of preparation. It first loops through every animation set in the actor and extracts its name and index. Each name is added to the top list box and the index of the animation set in the animation controller is assigned to the list box element as item data. When the user selects an item in this list box, we can extract the name they selected and the index of the animation set they would like to use as the new source set. This method will also initially disable the slider control because no source animation set will be selected initially. The slider is only used to scroll through the position of the currently selected source animation set and serves no purpose if a source animation set has not yet been selected.

When the PopulateOptions method returns, the top list box will be filled with all the animation set names currently managed by the actor. With the dialog controls now initialized, processing of this message is complete. We return true to let the OS know that we have handled this message and require no further default processing of this message.

In the next case, we test to see if the message passed is a WM_COMAND message. Many controls send WM_COMMAND messages. When a WM_COMMAND message has been sent to us by Windows, the

wParam parameter will contain vital information. In the low word of this variable will be stored the numerical ID assigned to the child control of the dialog (in the resource editor). This is the control that is sending the message and it usually means that the user has interacted or changed the contents of the control in some way. In the high word of wParam will be the actual event (notification message) that has occurred.

The following sections of code are all for the WM_COMMAND message case. In this first code block, we can see that if the WM_COMMAND has been received, we enter another switch statement to test which control sent the message. Since the control ID is stored in the low word of the lParam parameter, we can use the LOWORD macro to crack it out and examine it.

```
    case WM_COMMAND:

        switch ( LOWORD(wParam) )
        {
```

Now that we know we have a WM_COMMAND message, let us see what control sent it. The first test is for the control with the ID of IDC_LSTANIMSETS. This is the ID that was assigned to the top list box on the dialog box in the resource editor. It is the list box that will, at this point, contain the names of all animation sets currently registered with the animation controller. If the contents of LOWORD(wParam) matches this ID, then the user has performed some action in this list box which has triggered this message. For example, the user may have selected another animation set from the list. When this has happened, the list box control will always send a notification message to the parent window to notify it that a selection change has occurred.

Once we know it is the list box control that sent the message, we can test the high word of the wParam parameter for the notification code that was sent. The only event/notification we are interested in intercepting is the LBN_SELCHANGE list box notification message. Windows will send us this message when the user has changed the selection. In our example, it means the user has changed their choice of source animation set. You can see in the next section of code that if this is our list box that has sent the message, and if the notification it has sent is one of a selection change, we will call the CAnimDialog::CurrentSetChanged method. This method will extract the new selected animation set name and index from the list box. It will then use that set index to fetch the matching animation set interface from the actor and assign it to track 0 on the mixer. The track will be assigned an initial position of 0. This new animation set will now be the current animation set that will be manipulated by the slider control and will be the new source animation set for the split definition currently being constructed. Notice how we return true once we have processed the message so that Windows knows we have handled it.

```
        case IDC_LSTANIMSETS:
            if( LBN_SELCHANGE == HIWORD(wParam) )
             { CurrentSetChanged(); return TRUE; }
            break;
```

If the WM_COMMAND message was sent by a selection change in the bottom list box then the control sender ID will be IDC_LSTNEWANIMSETS. This is the list box which displays the currently compiled split definitions. If the user has selected one of these sets from the list, then as in the above case, a

selection change notification will be sent by that list box control. When a user selects an already added split definition from the definitions list, the details of the split definitions are displayed in the status window next to the list box. That is essentially the task of the CAnimDialog::NewSetChanged method. This helper function extracts the values from the selected definition structure and displays them in the status window.

```
        case IDC_LSTNEWANIMSETS:
            if( LBN_SELCHANGE==HIWORD(wParam) ) { NewSetChanged(); return TRUE; }
            break;
```

If the WM_COMMAND message sent was generated by the user pressing the Set Range Start or Set Range End buttons, then the IDs of the control sending the message will be IDC_BUTTON_SETSTART or IDC_BUTTON_SETEND, respectively. Once again, these are the IDs we assigned to these button controls in the resource editor. When this is the case, we call the SetRangeStart and SetRangeEnd methods. These methods simply extract the current position value from the slider control and store it in the m_nStartTick and m_nEndTick member variables depending on which of the two buttons was pressed. These values are cached in these member variables until the Add Definition button is pressed. At that time, the start and end marker values are copied into a new split definition structure in the dialog's definition array.

```
        case IDC_BUTTON_SETSTART:
            SetRangeStart();
            break;

        case IDC_BUTTON_SETEND:
            SetRangeEnd();
            break;
```

When the user has set the start and end markers for the current definition and wants to add that definition to the list, the Add Definition key is pressed. This button (whose ID is IDC_ADD_DEFINITION) sends a WM_COMMAND message. It is trapped in the next section of code and used to trigger the AddSetDefinition method. This method adds a new split definition to the end of the array. The previous cached start and end marker positions are copied into this structure and the method extracts the text the user has typed into the Set Name edit box and stores it in the split definition structure as the new animation set's name. There is also a Remove Definition button which, when pressed, triggers a call to the RemoveDefinition function. You can probably guess what this function does.

```
        case IDC_ADD_DEFINITION:
            AddSetDefinition();
            break;

        case IDC_DEL_DEFINITION:
            RemoveDefinition();
            break;
```

At some point, the user will have added all the desired split definitions to the dialog object's internal array of definition structures. At such a time, they will click the ApplyChanges button and trigger a WM_COMMAND message with the control ID of IDC_APPLY. As you can see in the following code

block, this will trigger a call to the CAnimDialog::ApplyChanges method. This method simply calls the CActor::ApplySplitDefinitions method and passes in the array of split definitions that has been compiled. When this function returns, the new animation sets will have been built and registered with the actor's animation controller and the old animation sets (the original source animation sets) will have been deleted. The following code shows the last sub-case for the WM_COMMAND message:

```
        case IDC_APPLY:
            ApplyChanges();
            break;

        } // End Control Switch
        break;
```

Not all controls send their notification messages on the back of WM_COMMAND messages. One such control is the slider control, which is not one of the original Windows controls. Instead, it belongs to a pool of controls referred to as the *common controls*. These controls were added after the core controls as a means of extending the user interface capabilities of the OS.

When a slider control is moved or changed it sends a WM_HSCROLL message (instead of a WM_COMMAND message) to the parent window. The lParam parameter will contain the *handle* of the control window that sent the message (i.e., the slider). However, what we actually want is the window ID (not its handle) just as had in the above cases, so that we can make sure that the ID of the control that sent this scroll message is indeed the ID we assigned to our animation set slider control in the resource editor.

Win32 has a function called GetDlgCtrlID which accepts a window handle and returns its control ID. In this next section of code, we are interested in scroll activity in our slider bar which has a control ID of IDC_ANIMSLIDER (assigned in the resource editor). When this is intercepted, the CAnimDialog::AnimSliderChanged method is called, which extracts the current position of the slider and uses it to update the position of track 0 on the animation controller. This is the track that the current source animation set is assigned to. This function will also force the actor to perform a hierarchy update pass so that when the CGameApp::FrameAdvance method renders the actor in the next iteration of the render loop, these changes are reflected in the output. We then return true from this function so that Windows knows we have handled this message.

```
    case WM_HSCROLL:

        switch ( GetDlgCtrlID( (HWND)lParam ) )
        {
        case IDC_ANIMSLIDER:

            AnimSliderChanged();

            // We handled it
            return TRUE;

        } // End Control Switch
        break;
```

```
    } // End Message Switch

    // Not Processed!
    return FALSE;
}
```

If the end of this function is reached, then it means that a control that we are not interested in sent the message, or a control that we are interested in sent a message of some type that we are not setup to handle. When this is the case, we return false so that Windows knows that we took no action and it can feel free to perform any default processing it may want to do. For example, we would want Windows to handle paint messages for controls when their contents need to be refreshed.

CAnimDialog::AnimSliderChanged

If we look at the source code to the CAnimDialog::AnimSliderChange function (called from the dialog procedure in response to a WM_HSCROLL message), we can see the typical interaction between the dialog box object, its controls, and how changes to those controls update the properties of the actor. This will enlighten you with respect to how the messaging system works. Remember, if your Win32 is a little rusty, controls and dialog box programming are all covered in Module II of the C++ Programming for Game Developers series here at the Game Institute.

The function first checks that a source animation set has been selected. If this is the case then the interface for that animation set will have been cached in the m_pCurrentAnimSet member variable. This interface pointer would have been extracted from the actor and stored in this member variable in response to the user selecting a new source animation set from the list (see the CAnimDialog::CurrentSetChanged method). Provided this is the case, we then need to get the window handle of the slider control using the GetDlgItem function. This function accepts, as its first parameter, the handle of the dialog box, and as the second parameter, the ID of the control we wish to get an HWND for. The ID of our slider control is IDC_ANIMSLIDER.

```
void CAnimDialog::AnimSliderChanged()
{
    HWND   hControl;
    TCHAR  Buffer[128];
    double Position, MaxSeconds;
    ULONG  TickPos = 0;

    // Break out of here if we don't have one selected atm
    if ( m_pCurrentAnimSet )
    {
        // Get the slider control hwnd
        if ( !(hControl = GetDlgItem( m_hDialog, IDC_ANIMSLIDER )) ) return;
```

If we get this far, then we have an HWND to the slider control stored in the hControl local variable. We then get the period of the currently select source animation set so we can make sure that the position they have provided is not out of the range.

```
        // Store max seconds variable
        MaxSeconds = m_pCurrentAnimSet->GetPeriod();
```

Next we need to get the position of the slider. When the animation set was first selected by the user in the CurrentSetChanged method, the period of the animation set would have been retrieved, converted to ticks, and then used to set the range of the slider control. This means that the slider control's range should be between 0 and MaxTicks at its start and end positions. We now need to extract the current position that the user has moved the slider to. We do this by sending the TBM_GETPOS message to the slider control using the Win32 SendMessage function. The function will return the current position of the slider (in ticks) so that we can set the current position of the track that the animation set is assigned to.

```
        // Retrieve the new animation set position in ticks
        TickPos = ::SendMessage( hControl, TBM_GETPOS, (WPARAM)0, (LPARAM)0 );
```

When we set the track position of the actor, we specify the new position in seconds, not ticks. So we must convert the new tick position we have just extracted from the slider control into seconds by dividing the tick position by the animation set's ticks per second ratio. We do not want the position to ever be greater than the period of the animation set; otherwise the periodic position of the animation set will loop back around to zero. Therefore, we calculate the new position as being the minimum of either the new track position extracted from the slider (in seconds) and the period (MaxSeconds) of the animation set. We never want the position to be greater than the period, so this test is done for safety. Notice we subtract epsilon from MaxSeconds to account for floating point errors and to make sure that the position is always clamped between the start and end of the animation set's period.

```
        // Convert to seconds for track (clamp - epsilon to prevent rendering loop
        //  around)
        Position = min( (double)TickPos /
                        m_pCurrentAnimSet->GetSourceTicksPerSecond(),
                        MaxSeconds - 1e-4 );
```

We now have the new position for track (always track 0 in this application) as described by the slider control. We use the CActor::SetTrackPosition method to apply this position change to the track. We then update the actor's absolute matrices by calling the AdvanceTime method. Notice how we pass 0.0 as the first parameter because we do not want to increment any track positions since we have just set the track position to the correct value. However we do pass in 'true' as the second parameter, which we know from our ealier discussion forces the actor to traverse the hierarchy and build its absolute marices for each frame. The next time the CGameApp::FrameAdvance method renders the actor, its matrices will have been updated and the new position of the actor (described by the new slider position) will be reflected in the render window.

```
        // Update track time
        m_pActor->SetTrackPosition( 0, Position );

        // Ensure all frame matrices are up to date
        m_pActor->AdvanceTime( 0.0f, true );

    } // End if animation set
```

At this point the actor has been updated, but we also have an edit box to update. Next to the slider is an edit box which displays the current position of the slider numerically for greater accuracy during the setting of the start and end marker positions. As the slider has now been updated, we should also replace the text in this edit box with the new position as well.

We do this by using the GetDlgItem method to retrieve the HWND for the edit window. The ID of this edit window is defined in the dialog template as IDC_EDIT_CURRENTPOS. Once we have the window handle, we convert the tick position that we just extracted from the slider into a string stored in a local char array (Buffer). We then change the text of the edit window to the value stored in this string using the Edit_SetText macro (defined in windowsx.h).

```
    // Get the edit control
    if ( !(hControl = GetDlgItem( m_hDialog, IDC_EDIT_CURRENTPOS )) ) return;

    // Get a string of the position in ticks
    _itot( TickPos, Buffer, 10 );

    // Set the value
    Edit_SetText( hControl, Buffer );
}
```

Note: The Edit_SetText macro is just a friendly wrapper around the SendMessage function used to send messages to all windows. All the core controls (list boxes, edit boxes, buttons, etc.) have macros defined for them just like this one in windowsx.h. They make the setting and getting of control properties easier and more user-friendly. Notice however that we do not use a macro like this when getting the position of the slider control. This is because the slider control is not one of the original core controls of the operating system and therefore has none of these macros defined for it. We have to do it the slightly more involved way for the slider (and any common control) by making the call to the SendMessage function ourselves. All the Edit_SetText macro is doing is wrapping the sending of the WM_SETTEXT message to the edit control using the SendMessage function.

CAnimDialog::CurrentSetChanged

CurrentSetChanged is called from the dialog box procedure in response to the user selecting a new item in the top list box on the dialog. This means the user has selected a new source animation set that it wishes to split.

The first thing we do is retrieve the HWND of the list box that the animation set names (and indices) are stored in. We use the GetDlgItem function again to retrieve the HWND of the list box with the control ID of IDC_LSTANIMSETS. We then use the ListBox_GetCurSel macro (see windowsx.h) to retrieve the current item that is selected in the list box as an integer index. Remember, this function was called because the user selected a new item. It is the index of this newly selected item that is returned from this function.

```
void CAnimDialog::CurrentSetChanged()
{
    HWND                        hControl = NULL;
    LPD3DXANIMATIONSET          pAnimSet = NULL;
    LPD3DXKEYFRAMEDANIMATIONSET pKeyAnimSet = NULL;
```

```
    // Get the current selection
    if ( !(hControl = GetDlgItem( m_hDialog, IDC_LSTANIMSETS )) ) return;
    ULONG nIndex = ListBox_GetCurSel( hControl );
```

We now know the index of the item in the list box that the user has just selected and we can use it to fetch the corresponding animation set from the actor and cache it in the member variables of the CAnimDialog object. First, let us release any current animation set information that may be cached from a previous selection.

```
   // Release the previously selected animation set
    if ( m_pCurrentAnimSet ) m_pCurrentAnimSet->Release();
    m_pCurrentAnimSet = NULL;
    m_nCurrentAnimSet = 0;
```

Next we test that we did get a valid selection index from the list box. If not, then it means the user has deselected an item, leaving no item selected. When this is the case, we call the CAnimDialog::EnableSetControls method, which disables all the slider and marker controls. They have no meaning if a source animation set is not selected. Also, if no animation set is selected, we make sure we also remove any previous current animation set from the first track on the mixer.

```
    // No item / error ?
    if ( nIndex == LB_ERR )
    {
        // Disable the animation set controls
        EnableSetControls( false );

        // Actor uses no set atm.
        m_pActor->SetTrackAnimationSet( 0, NULL );

    } // End if nothing selected
```

If a valid index was returned, then it is time to find out which animation set this index maps to in the actor. We can then fetch the interface of this animation set from the actor and assign it to track 0 on the mixer. Notice how each string in the list box also has assigned (as user data) the index of the animation set in the controller that the item maps to. Therefore, we use the ListBox_GetItemData to retrieve the index of the set we need to fetch from the animation controller.

```
    else
    {
        // Retrieve the item data for this
        ULONG nAnimSet = ListBox_GetItemData( hControl, nIndex );

        // We're now using this animation set
        pAnimSet = m_pActor->GetAnimationSet( nAnimSet );

        // We can only split up keyframe animation sets. Ensure this is available
        if ( FAILED(pAnimSet->QueryInterface( IID_ID3DXKeyframedAnimationSet,
                                              (LPVOID*)&pKeyAnimSet )) )
        {
            // Disable the animation set controls
            EnableSetControls( false );
```

```
            // Actor uses no set.
            m_pActor->SetTrackAnimationSet( 0, NULL );

            // We're done
            return;

        } // End if not keyframed

        // We're done with the normal animation set
        pAnimSet->Release();
```

After fetching the animation set, we make sure it is a keyframed animation set, since those are the only ones we know how to subdivide in this tool. If the selected animation set is not a keyframed set then we once again disable the marker and slider controls and clear track 0 on the animation mixer.

If we get this far, then the user has selected a valid source animation set. We set the animation set to track 0 on the mixer (ready for playback) and cache the animation set interface and its index in the member variables of the dialog object. We then enable the marker and slider controls so that the user can begin to use the slider bar and set their markers. Until an animation set is selected, these controls are ghosted and cannot be used.

```
        // We're using the keyframed animation set
        m_pActor->SetTrackAnimationSet( 0, pKeyAnimSet );

        // Store the key anim set, and don't release this one!
        m_pCurrentAnimSet = pKeyAnimSet;

        // Store the index too for later
        m_nCurrentAnimSet = nAnimSet;

        // Enable the animation set controls
        EnableSetControls( true );
```

There is one final job to do before we leave this function. When configuring the properties of a split definition, we use the slider to set two markers. We also have to assign that split definition a name using the Set Name edit box. When the user changes the source set, we will copy into this edit box the name of the source set by default. The user can change it, and will most likely want to, but if they are simply trimming an animation set, then this saves them from having to re-type the name of the new animation set when they wish to keep it the same as the original.

So before we exit this function, we fetch the name from the animation set currently selected and copy it into a temporary buffer. If the animation set has no name then we will simply fill this buffer with the string “Unnamed Animation Set”. We then get the handle of the Set Name edit control and set its text to the contents of this buffer.

```
        // Set the text for the new set name
        TCHAR   NameBuffer[512];
        LPCTSTR strName = pKeyAnimSet->GetName( );
        if ( strName == NULL || _tcslen( strName ) == 0 )
            _tcscpy( NameBuffer, _T("Unnamed Animation Set") );
```

```
        else
            _tcscpy( NameBuffer, strName );

        // Get the definition information box
        if ( !(hControl = GetDlgItem( m_hDialog, IDC_EDIT_NEWSETNAME )) ) return;
        Edit_SetText( hControl, NameBuffer );

    } // End if selected a legitimate item
}
```

CAnimDialog::AddSetDefinition

This method is called after the user has positioned the start and end markers, set the new name, and wants to add this new definition data to the list of set definitions. The first section of code just makes sure that the user does not try to add a large number of definitions. It prevents overflow of the dialog's internal array of animations which is allocated at application startup to be MAX_DEFINITIONS in size. This is set to 256 by default. This means that this tool allows you to break a single set into 256 sets. Feel free to change this value in the source code if you need more.

```
void CAnimDialog::AddSetDefinition()
{
    HWND  hControl = NULL;
    TCHAR NewName[128];
    ULONG i;

    // Validate there are enough slots left
    if ( m_nDefCount >= MAX_DEFINITIONS )
    {
        ::MessageBox( m_hDialog, _T("You have reached the maximum number of
                                    animation set definitions supported by
                                    this application."),
                                  _T("Limit Reached"),
                                  MB_OK | MB_ICONSTOP );
        return;

    } // End if limit reached
```

This next section of code tests that a source animation set is currently selected since each split definition must contain a source animation set. If a source animation set is not selected when the user presses the Add Definition button, then this split definition should fail to be added to the list. This should never actually happen because those controls should be disabled if no animation set is selected, but let us be cautious about this.

```
    // Validate that an animation set is available
    if ( !m_pCurrentAnimSet )
    {
        ::MessageBox( m_hDialog, _T("You must specify a source animation set to
                                    extract data from."), _T("Invalid Source"),
                                    MB_OK | MB_ICONEXCLAMATION );
        return;
    } // End if no source
```

We will also fail if the start and end markers are equal. This would essentially cause the creation of an empty animation set, which would serve no prupose.

```
    // Validate that some range of data is selected
    if ( (m_nEndTick - m_nStartTick) == 0 )
    {
        ::MessageBox( m_hDialog, _T("The data range specified is empty,
                                    you must select a portion of the animation
                                    to extract."), _T("Invalid Range"),
                                    MB_OK | MB_ICONEXCLAMATION );
        return;

    } // End if no source
```

Next we make sure that the user has supplied a name for the definition. We want to enforce this behavior so that all of our new animation sets will have unique names. This is especially important since they are displayed in the list boxes using their names. So we fetch the handle of the Set Name edit control and extract its text. If the length of the retrieved text is zero, the edit box is empty and we fail to add the definition to the list.

```
    // Get the new set name
    if ( !(hControl = GetDlgItem( m_hDialog, IDC_EDIT_NEWSETNAME )) ) return;
    Edit_GetText( hControl, NewName, 128 );

    // Empty ?
    if ( _tcslen( NewName ) == 0 )
    {
        ::MessageBox( m_hDialog, _T("All new set definitions must have a name."),
                      _T("Invalid Option"), MB_OK | MB_ICONEXCLAMATION );
        return;

    } // End if no name
```

When working with text mode X files, there are certain characters reserved by the specification for the purposes of structuring the data. These characters include semi-colons and curly braces. In certain circumstances, when we include such reserved characters in animation set names, we can actually inadvertantly corrupt the integrity of the file.

For this reason, we introduce the following loop, which forcably removes any non-alphanumeric characters from the animation set name and replaces them with an underscore. This prevents the problem from occuring.

```
    // Replace all non numeric / alpha characters with underscores
    // (the text .X file definition doesn't seem to like much else)
    for ( i = 0; i < _tcslen( NewName ); ++i )
    {
        TCHAR ch = NewName[i];
        if ( (ch < _T('A') || ch > _T('Z') ) &&
             (ch < _T('a') || ch > _T('z') ) &&
             (ch < _T('0') || ch > _T('9') ) )
        {
            // Replace
            NewName[i] = _T('_');
```

```
        } // End if invalid character
    } // Next Character
```

Now we are almost ready to add the definition information to the array. However, we must make sure that a split definition does not already exist with the same name. We enforce (for the same reasons mentioned above) the rule that all definitions must have unique names. In the next section of code, we loop through each split definition in the array and compare its name with the name of the new definition we are about to add. If the _tcscmp function returns zero, the strings are the same and we have found that a split definition already exists with the same name as the one we are about to add to the list. When this is the case, we fail to add the split definition.

```
    // Search to see if a name matching this already exists
    for ( i = 0; i < m_nDefCount; ++i )
    {
        if ( _tcsicmp( m_pDefList[i].SetName, NewName ) == 0 )
        {
            ::MessageBox( m_hDialog, _T("A new set definition with this name
                                        already exists in the list. Names must
                                        be unique."),
                                        _T("Invalid Option"),
                                        MB_OK | MB_ICONEXCLAMATION );
            return;

        } // End if matching name found

    } // Next Definition
```

If we get this far, the marker positions, the source animation set, and the name we have assigned to this new set are all valid, so we will copy the information into the array. We already have the new name stored in a local variable and the currently selected source animation set index was also cached in the member variable when the the user selected the animation set (see previous function). Also, when the user selects the Set Range Start and Set Range End buttons, the handler functions cache the current position of the slider in the m_nStartTick and m_nEndTick member variables. Therefore, the only thing left to do is add the information to the end of the definitions arrays and increase the definition count. We then call the PopulateDefinitions function to refresh the split definitions list box so that the new addition is now shown.

```
    // Populate the new definition item
    _tcscpy( m_pDefList[ m_nDefCount ].SetName, NewName );
    m_pDefList[ m_nDefCount ].SourceSet  = m_nCurrentAnimSet;
    m_pDefList[ m_nDefCount ].StartTicks = m_nStartTick;
    m_pDefList[ m_nDefCount ].EndTicks   = m_nEndTick;

    // Increment
    m_nDefCount++;

    // Repopulate Definition Items
    PopulateDefinitions();
}
```

CAnimDialog::ApplyChanges

The final method that we will examine in this lab project is the one that actually creates the new animation sets from the split definitions list. Although this might seem like it would be a large function, it actually just calls the CActor::ApplySplitDefinitions method to do all the work.

The function returns if no valid actor is available or if no definitions have yet been added by the user.

```
void CAnimDialog::ApplyChanges()
{
    // Bail if there is no actor
    if ( !m_pActor ) return;

    // Ensure there are any definitions to apply
    if ( m_nDefCount == 0 )
    {
        ::MessageBox( m_hDialog, _T("There are no definition changes to be
                                     applied."),
                                    _T("Empty Definition"),
                                    MB_OK | MB_ICONEXCLAMATION );
        return;

    } // End if no name
```

Now we know we have a valid actor and at least some definitions in our array, so we will release the currently selected animation set interface. This is because this animation set is about to be destroyed and replaced by the new ones described by the split definitions. We then call our CActor member function to create the new sets and release the old ones.

```
    // Release the currently selected animation set, it's about to be destroyed.
    if ( m_pCurrentAnimSet ) m_pCurrentAnimSet->Release();
    m_pCurrentAnimSet = NULL;
    m_nCurrentAnimSet = 0;

    // Apply the split definition
    m_pActor->ApplySplitDefinitions( m_nDefCount, m_pDefList );

    // Total re-population please
    UpdateDialog();
}
```

Finally, at the end of the function we call the CAnimDialog::UpdateDialog method to refresh all the controls with their new information. This will empty the definitions array and repopulate the top list box with the new list of source animation sets (the ones we have just created). We now have a new list of source sets and an empty array of definitions (as they have all be used and discarded). Usually, at this point you will want to save the actor back out to an X file in its new format.

Conclusion

This workbook has been one of our largest to date. In the process, we have learned alot of new material and have simultaneously turned our CActor class into a powerful tool. It now encapsulates X file loading and saving for complete multi-mesh hierarchies that include animation data. We have also exposed the powerful features of the D3DX animation controller through our CActor interface. Finally, in Lab Project 10.2 we developed a utility that better educates us about how the actor might be used in a non-game application. It is also a tool that will be extremely useful for breaking an animation set with combined sequences into multiple animations sets for the purposes of our game engine.

In the next two chapters we will extend the actor even further when we learn how to perform character skinning using both software and hardware techniques. Make sure that you are comfortable with everything we have covered thus far, as it will all be used again in the next few lessons.