Chapter Ten

Animation

Introduction

To date our coursework has focused primarily on the three dimensions of physical space. We looked at how to build, position, and orient models in a 3D world so that we can take a snapshot of the scene and present it to our player. But there is also a fourth important dimension that we have looked at only briefly: time. Time is the dimension introduced when animation becomes a part of our game design.

To animate something means literally to "give life" to it. This is clearly an important concept in computer games. Of course, most of our lab projects have included animation in one form or another. From the spinning cubes back in Chapter One to the movement of our player/camera through the world as the result of user input, we have had a degree of animation in almost all of our demos. In this lesson we will look at some of the more traditional animation concepts that one thinks about when working with 3D computer graphics.

In today's 3D computer games, being able to animate 3D models and scenes in a controlled or pre-recorded way is absolutely essential. We will see later in the course how animated characters in computer games will have many pre-recorded animations that our application can select and playback at runtime in response to some game event. For example, a character might include pre-recorded animations for walking and running so that as it travels from point A to point B in the world, it looks correct while doing so. There may also exist pre-recorded animations of several death sequences for the character that can be played back by our application in response to the character's health falling below zero. We would also need animations to play when character itself is firing its weapon, etc.

Most of the difficult work of building and capturing animation data is the domain of the project artist, modeller, and animator. Most popular 3D modelling packages include tools to animate 3D models and export that data. As programmers, our job will focus on integrating the results of their efforts into our game engine. Our only real requirement is that the data be delivered in a format supported by the engine we have designed. In this particular lesson, that format will be the X file format. Fortunately, most commercial modelling programs support the export of animated scenes to the X format, either natively or through the use of a plug-in. Those of you who are fortunate to own or have access to high-end programs like 3D Studio Max™ and Maya™ will indeed have a great opportunity to design and export complex animations. There are also excellent packages like Milkshape3D™ that are a more cost-effective solution for the hobbyist, that still offer a wealth of animation and export options.

> **Note:** Our focus in this lesson will be on working with pre-recorded animation sequences that are created offline by our project artist. However, it should be noted that there are also animation techniques that can be calculated procedurally at runtime to produce certain interesting behaviours. These techniques (such as inverse kinematics to cite just one example) will be explored later in this course series.

Real-time animation is such an integral part of today's videogame experience. Not so many years ago, video games would typically play pre-recorded movie clips called 'cut-scenes' at key points in the game to help move the storyline forward. While the quality of those cut-scenes generally exceeded the in-game graphics quality, there were a number of criticisms that followed from their use. First, they would often be associated with large load times which would destroy the sense of immersion and game flow.

Perhaps more importantly, the player was transformed from being a participant in a 3D world where they had total freedom of movement, to passively watching non-interactive slide shows or movies.

In modern games, given that the hardware they run on has more dedicated processing power for graphics and animation, cut-scenes are commonly played out using the actual game engine. This allows them to become an integral part of the game experience, where in-game story events can happen on the fly without the need to cut away and make the player idly watch. Player immersion is preserved and the world feels much more real. Players can often continue to move about the world even while these scripted events are playing out. If you load in Lab Project 10.1 you will see a very simple pre-scripted animation of a spaceship taking off and flying into the distance. While this is the simplest of examples as far as such things go (normally such events are triggered as part of an evolving storyline), the point is that you are still allowed to move freely around the world while the sequence plays out around you. Imagine that just prior to the ship taking off, your squad mate informed you that he would commandeer the ship and meet you at the rendezvous point. Then you watched him jog off into the distance under heavy fire, board the ship, and take off for deep space. All the while, you were laying down covering fire for him as he made his way across the dangerous killing field. While our demo is not nearly as exciting, you can certainly imagine such a sequence taking place in a commercial game. Modern 3D game titles are littered with these small but often critical sets of scripts and animations and they are vital to making the 3D world feel more realistic. But even when this is not the case and a game actually cuts away and the player must simply passively watch the sequence, using the in-game graphics engine provides a sense of consistency that keeps the player immersed in the world.

Finally, it is worth noting that it is much more efficient to store scripts and animations for the game engine (the animations are generally built from simple translation and rotation instructions that can be interpolated -- more on this shortly) than it is to store full motion video sequences at 30 frames per second. Such videos can consume significant disk space, and require important processor time especially if used frequently.

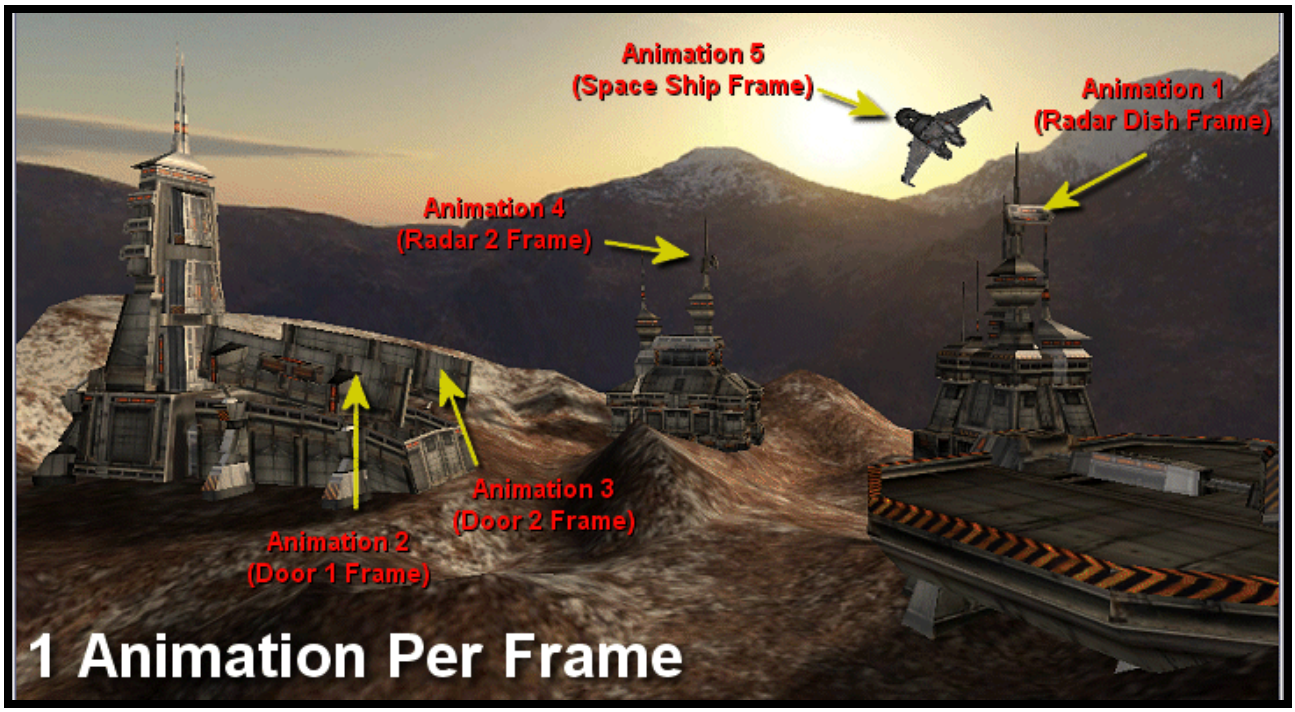
10.1 Basic Animation

Before we examine the underlying mechanisms of the DirectX animation system, it will be encouraging to take a quick look at how easy it is to load a hierarchical X file that includes animation data and then play that animation in our game engine. D3DXLoadMeshHierarchyFromX allows us to load hierarchical X files that include animation, so we are already on the right track. The final parameter we pass to the function is the address of a pointer to an ID3DXAnimationController. If the file contains animation data, then this pointer will ultimately be used to access, manipulate, and play those stored sequences. If the file contains no animation data, then this pointer will be set to NULL on function return. With this in mind, let us see how we would load an animated X file. In many ways this next code snippet brings together everything that we have learned so far:

```
CAllocator Allocator;
D3DXFRAME * pRootFrame = NULL;
ID3DXAnimationController * pAnimController = NULL;
D3DXLoadMeshHierarchyFromX( "MyScene.x", D3DXMESH_MANAGED, pD3DDevice, &Allocator,
                            NULL, &pFrameRoot, &pAnimController );
```

The animation data stored in the X file will be used to manipulate the values stored in the hierarchy frame matrices. When loaded by D3DX, each frame in the hierarchy that the artist decides to animate will have attached an **Animation Data Set** that contains a set of animation information for a single frame in the hierarchy. You should recall our earlier theoretical discussion about frame animation in the Chapter Nine -- our CAnimation object simply applied a rotation to the frame to which it was attached. But we are already familiar with the concept of these objects being attached to frames in the hierarchy and being responsible for managing their animation. While D3DX does not create a separate object to contain the animation data for a single frame in the hierarchy, we can think of the animation data for each frame as being a packet of animation data that is assigned to an animation set and registered with the animation controller. For example, if the file contains a single animation set that animated 10 frames in the hierarchy, D3DX would create an animation set object that contained 10 packets of animation data, one for each frame. Each packet is also assigned a name that matches the name of the frame to which it is assigned and to which its animation data applies. This name is also referred to as the name of the Animation. Therefore, in DirectX terminology, when we discuss an 'Animation' we are referring to a set of animation information linked to a single frame in the hierarchy and contained inside an animation set.

Lab Project 10.1 provides a basic demonstration of animating a scene hierarchy. It includes a small animated spacecraft, a pair of hanger doors that open and close, and rotating radar domes on two of the buildings. The entire scene is stored as a single hierarchy, but only certain frames include attached animations. There are in fact, five animations in total: one for each animated frame in the scene. Static frames do not have animations created for them. The five animations in this scene were all created by D3DXLoadMeshHierarchyFromX automatically when we loaded the X file. An animation set was created and the five animations were added to it. The animation set was then registered with the animation controller so that on function return, our application can start playing the animation(s) immediately.



At a high level, the animation controller returned from the D3DXLoadMeshHierarchyFromX function is like an **Animation Set Manager**; it stores all of the animations found in the X file (in one or more animation sets) and manages communication between them and the application. When the application wants to advance the global timeline of the animated scene, we instruct the animation controller via its AdvanceTime method. The animation controller in turn relays this timeline update to each of its currently active animation sets. Each animation set is a collection of frame animations. Each frame animation stored in the animation set is used to generate the new matrix for the frame to which it is attached for that position in the timeline. In short, with a single call to the ID3DXAnimationController::AdvanceTime function, all active animation sets update the frame matrices for which they have animations defined.

> **Note:** While the first part of this lesson will focus on populating the D3DX animation system with animation data stored in X files, we will see later that D3DX provides the ability to construct all the frame interpolators, animation sets and the animation controller manually. This would allow an application to import data from any file format into the D3DX animation system. This would be done by manually attaching animations to the various frames in the hierarchy and registering them with animation sets on the animation controller. We will discuss how to manually build the components of the D3DX animation system later.

While the techniques for creating simple animated scenes vary from application to application, one concept remains universal: **the animation timeline**. Every animation package includes a timeline where transformations or other types of events can be placed in chronological order. These events are called *keyframes* and they represent the most important transitions that must take place at particular moments in time. Time itself can be measured in seconds or milliseconds or by the more arbitrary timing method of 'ticks'. When D3DX loads animation sets from an X file, every animation set will include its own animation timeline. Therefore, while our application will be advancing a single global time via the ID3DXAnimationController::AdvanceTime method, each animation set has its own local timeline which the animation controller takes care of updating. The length of an animation set's timeline (also referred to as its *period*) is equal to the length of its largest defined **animation**. Remember that an **animation** is a set of animation data for a single frame in the hierarchy.

> **Note:** As the D3DX loading functions load their animation data from the X file format, it is important that you install the appropriate 'X file Exporter Plug-in' for your chosen 3D modelling application. 3D Studio Max™ is a very popular choice for artists and modellers, so if this is your chosen application then you are in luck since there are a couple to choose from. While the DirectX9 SDK ships with an 'X File Exporter' plug-in for 3D Studio Max™, you are provided with only the source code. This means you must compile the plug-in yourself, which comes with its own set of problems as discussed in Chapter 8. Luckily, there are some very good 3rd party X file exporters for MAX (one called 'Panda X File Exporter' which is freely available is worth investigating). There are also exporter plug-ins available for other popular programs, such as Maya™ and Milkshape™ for example.

Each animation set will have been defined by the artist using its own local time. As several animation sets may be playing at once, each of which may have different durations (periods), the ID3DXAnimationController::AdvanceTime is used to calculate and mix the local animation timelines using a single global time. For example, Animation Set 1 may have a duration of 10 seconds, while Animation Set 2 may have a duration of 30 seconds. If both of the animation sets were created to be looping animations (more on this in a moment), then their timelines will wrap back around again to the beginning when the global time of the animation controller exceeds their local time. If the global time of the animation controller was at 35 seconds, at which periodic position on their local timelines would

each of the above animation sets be? Animation Set 1 would be 5 seconds into its local timeline and Animation Set 2 would also be 4 seconds in. Why? Consider the wrapping property of the animation sets. At 35 seconds, Animation Set 1 (which only lasts of 10 seconds) would have wrapped around three times and would now be on the fifth second of its fourth payback. Animation Set 2 which lasts for 30 seconds reached the end of its timeline 5 seconds ago, so now has a local periodic position of 5 seconds into its second playback.

Updating the animation sets and instructing them to rebuild their associated frame matrices is actually quite simple. The ID3DXAnimationController interface includes a method called AdvanceTime which serves as the main update function for the animation controller and all its underlying animation sets. The application will pass in a value in seconds that represents the *elapsed* time (i.e., the time since the last call to AdvanceTime) that we would like to use to progress the animation controller's global timeline. This value is passed on to each animation set in the scene so that their local timelines are updated, and wrapped if necessary. After an animation set timeline update, the animation set will loop through each of its stored animations and use their data to re-build the frame matrices assigned to them from the hierarchy.

The following code shows a simple game loop that cycles the animation:

```
void GameLoop( float ElapsedTime )
{
      pAnimController->AdvanceTime( ElapsedTime );
      RenderFrame( pRootFrame, NULL );
}
```

The GameLoop function would be called every frame and is passed the amount of time (in seconds or fractions of a second) that has passed since the last render. The call to ID3DXAnimationController::AdvanceTime updates the controller's global time and also passes the elapsed time along to each of its animation sets so that their local timelines are also advanced. Each animation set will use this new local time to perform a look up in its list of stored animations. Each animation (which stores animation keyframes for a single frame in the hierarchy) will be used to calculate what the matrix for the assigned frame should look like at the specified time. When the function returns, all of the animations stored in the animation set will have updated the matrices of their attached frames in the hierarchy.

> **Note:** It is important to emphasize that time drives the animation loop. This is critical to maintaining smooth animation independent of frame rate. While it might seem easier to simply increment a frame counter for controlling animation, it is much more risky given the different speeds your game will run on different hardware. Locking the frame rate can minimize the problems, but that is not usually a desirable method as it imposes a limitation you probably do not want. Nor can you count on all hardware even meeting your minimums. Since the animation sequences in modelling packages are based on time, the engine should be as well. This is certainly easy enough to accommodate, so do not try to cut corners here.

Now all we have to do is render the hierarchy. As we saw earlier, this involves stepping through the hierarchy, combining the relative matrices as we go. Any of these relative frame matrices that have had animations assigned (in an active animation set) will have been updated at this point, so the changes will automatically affect all child frames and meshes. For example, if we were to apply a rotation animation

to the root frame, although there would be only a single animation in the animation set currently being played, and the AdvanceTime method would only affect one matrix directly (the root matrix), the entire frame hierarchy will rotate because all child matrices are relative transformations. When we build the world matrices for each frame during the traversal and matrix concatenation process (i.e., the update pass), these changes would be reflected down through all children. So essentially, by advancing the global animation time via the animation controller, we are simply instructing all contained animations to update their attached frame matrices to reflect what the position and orientation of the frame should be at this time. Remember, it is the relative frame matrices that are updated -- the ones that were loaded from the X file. We still have to generate the world matrices for each frame using the standard traversal and concatenation technique, after these relative matrices have been updated.

As discussed earlier, it is often preferable to separate our rendering logic into an update pass and a drawing pass. Recall that this involved maintaining a local copy of the world matrix for each frame that is updated as the hierarchy is traversed. We can modify our previous GameLoop function to accommodate this strategy very quickly:

```
void GameLoop( float ElapsedTime )
{
      pAnimController->AdvanceTime( ElapsedTime );
      UpdateHierarchy( pRootFrame, NULL );
      RenderHierarchy( pRootFrame, pCamera );
}
```

It is worth noting that the example GameLoop function above does not bother wrapping around the global animation time value but simply continues to advance it. As it happens, if the animation sets have been created such that they should loop, it will automatically take any value that is out of bounds and turn it into a time within its animation set's timeline based on the length of the animation set. For example, if we have an animation set that runs for 15 seconds (i.e., its longest animation runs for 15 seconds) and the global time of the animation was currently at 18 seconds, it would be treated as 3 seconds into the second animation loop. A value of 31 would be treated as 1 second into the third loop of the animation, and so on. If your animation sets are not configured to loop then you might not want to continue increasing the global time of the animation controller. Instead you might want to reset it back to zero when it gets too large. This will allow the non-looping animations to play again when the global time restarts at zero.

The DirectX animation system underwent significant change in the 2003 summer update of the SDK. Previously, individual animations (formerly called Interpolators) within an animation set also had their own local timelines. This third time layer was extremely useful as it meant that you could have several animations within an animation set, all with different durations and independent looping capability. This was arguably more convenient, but perhaps this third layer of individual animation independence was slowing down the entire system. This could potentially justify its demise. However, because individual animations within an animation set are now just 'dumb' animation data containers for frames in the hierarchy, the artist will often have to make sure that all animations assigned to a given animation set last for the same duration. This is because the animation set's local timeline only wraps around when the end of its period is reached (its period is the duration of its longest running animation). When the 2003 summer update was released, the assets for Lab Project 10.1 had to be completely rebuilt. Initially, the five animations in our space ship scene were all part of the same animation set. The radar dish

animations only lasted a short period of time compared to the length of time it took the spaceship to take off and fly away into the distance. We can see from watching the scene that the radar dishes will rotate many times during this sequence. Previously, it did not matter that the radar dish animation did not last as long as the space ship animation because although they belonged to the same animation set, the animations themselves had their own loop-capable timelines. Once the radar dish had finished one complete cycle, it would just wrap around and do it again. This support has now been removed from DirectX 9 and there is no per-animation timeline concept anymore. To upgrade our assets, we had to insert extra animation data for the radar dishes (adding extra rotations implicitly) so that the duration of every animation lasted as long as the flying ship. All animations now have the same duration, which is equal to the duration of the animation set to which the animations belong. One might argue that this was not exactly a change for the better, but so it goes.

So at its simplest, rendering an animated X file scene involves only one additional function call (ID3DXAnimationController::AdvanceTime). Very often, this may be all your application needs to do. Of course, the D3DX animation subsystem is much more sophisticated than this and includes many additional interfaces and function calls. But we can see now that including pre-generated animations in our existing game engine is actually quite an easy thing to do.

The remainder of our chapter will cover the D3DX animation interfaces and the functionality they expose. We will soon see that the ID3DXAnimationController has much more depth than we saw in this section. While it can be used in the very simple way we just learned, where the entire scene was animated using just one function call, it can also be used to produce animation techniques of great complexity.

10.2 Keyframe Interpolation

Animation is a time-driven concept. We know that each frame in our hierarchy can be assigned its own animation which maintains a timeline of animation information for that frame. Along the timeline at specified intervals are keyframes which store transformation information such as position, rotation, and scale. This is a very efficient way to store animation. By interpolating the transformation data stored in the keyframes at runtime, we are able to dynamically generate our animation data (a frame matrix) for any given moment in the game. This means that we will not need to store a copy of the scene data in every possible position/orientation along the timeline; we simply need a handful of stored keyframes that represent the important 'poses' along the timeline. This saves a lot of storage space, and certainly makes the artist's life much easier. Animation data is stored in an X file (and in most modelling files) as keyframes.

> **Note:** Understanding how keyframe animation works is simpler if you know how to create the data in a modelling package. A small tutorial on how to create animation data in a 3D modelling package can be found in Appendix A. It would be helpful if you read this short tutorial before continuing.

Let us now spend a little time discussing keyframes in more detail and how the D3DX animation controller will use them to generate animation data at runtime.

One of the earliest forms of animation was the flipbook. A flipbook contains a collection of pages, each with a drawing that is slightly altered from the previous page, such that when we rapidly flip the pages from front to back or vice versa, the drawing appears to animate.

Let us imagine a 100 page flipbook containing an animation of a ball rolling along the ground from left to right. On page 1 the ball is at its furthest point left, and on page 100 the ball has been drawn at its far right position. For all pages in between, the ball is drawn such that it moves further to the right as the page number increases.

If someone were to ask you to guess the position of the ball on an arbitrary page in the book, your guess would probably be reasonably correct. On page 50, you would know that it should be equidistant from its start and end positions. On page 25, it would be about 25% of the way from its start position with about 75% of the path left to traverse. We can make these assumptions because we know in advance that the animation is designed to represent a linear translation between two known locations over a known number of pages (assuming a constant speed of course).

Now imagine that we rip out all of the pages except the first and last pages. Do you think that you could fill in the images on the 98 missing pages by hand? Of course you could. This is because you know where the ball needs to be on any given page.

Keyframes are essentially like the first and last pages in the flipbook. They tell us the state of an object at a given point in an animation sequence. Once we know the starting state and ending state, it just becomes a case of filling in the gaps.

Let us think of this example in the context of our animation controller. Our hierarchy will contain a single frame (call it BallFrame) with an attached animation. The animation would store two keyframes. The first keyframe would describe the starting position of the ball frame and the time at which the ball should be placed at this position. Since this is the start keyframe, the time will be zero. The second keyframe would contain the end position for the ball and the time at which the ball should reach this position. Let us say for example that animation will last 20 seconds. So keyframe 2 will have a timestamp of 20.

> **Note:** This is a simple example for the purposes of explaining the concept. A keyframe's timestamp is not normally stored explicitly as seconds, but is usually stored using a measurement of time with a higher granularity. This can actually be an arbitrary time unit as we will see in a moment. Either way, D3DX will take care of loading this data in so that we can specify our time in seconds in the 'AdvanceTime' method of the ID3DXAnimationController interface.

Now, let us say we were to call ID3DXAnimationController::AdvanceTime with a value that would advance the global timeline of the controller to 15 seconds. Let us also assume that the animation set's local timeline is also set at 15 seconds. For each animation contained within an animation set, we can locate the two keyframes that most closely bound the passed time (i.e. the closest keyframes to the left and right of the current time on the timeline). In this simple example, we will say there are only two keyframes describing a position at 0 seconds and a position at 20 seconds. These are the start and end positions of our 'rolling ball' flipbook analogy. We know that in this case the two keyframes that are closest to either side of the passed time (15 secs) will be the start keyframe (0 secs) and the end keyframe (20 secs).

With the bounding frames located, the animation set will now use the passed time to linearly interpolate between the translation values stored in the keyframes. This will determine exactly where the ball should be at 15 seconds. Next we see a very simple example of what the animation set might do to calculate the position of an animation called 'Ball_Animation' at 15 seconds. For reasons of clarity we are referencing the animation data for 'Ball_Animation' as if it is a C++ object. In reality, it would just be an array of keyframes stored in the animation set assigned the name 'Ball_Frame'.

```
PassedTime = 15.0;

D3DXVECTOR3 StartPos = Ball_Animation.KeyFrame[0].Position;  // Position at 0 seconds
D3DXVECTOR3 EndPos   = Ball_Animation.KeyFrame[1].Position;  // Position at 20 seconds

float s  =  PassedTime - Ball_Animation.KeyFrame[0].Time;
      s /= ( Ball_Animation.KeyFrame[1].Time - Ball_Animation.KeyFrame[0]Time);

CurrentPosition = StartPos + (s * (EndPos - StartPos));
```

In reality, when the current position is calculated, it would be placed inside the frame matrix that the ball animation is attached to. We do not have to implement any of this code, as it is all handled by the D3DXAnimationSet object contained inside the animation controller. We are currently just trying to get a better feel for how the whole system works.

What if we wanted the ball in our example to jump up in the air somewhere around the middle of the animation? Not a problem. More complex animations simply require more key-frames, every subsequent pair of which describes a single transition between events. To create the jump effect, we could use the keyframes seen in Fig 9.6.

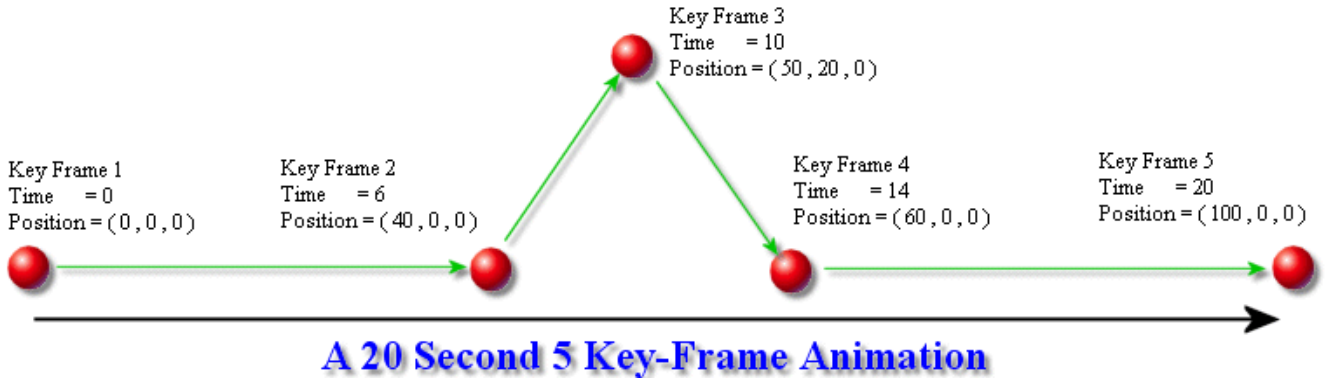


Figure 9.6

Using the example in Fig 9.6, let us think about what will happen if we passed in an elapsed time of 8 seconds.

First the animation set would find the two bounding keyframes for this time. Since we passed in a time of 8 seconds, the two bounding keyframes would be keyframe 2 and keyframe 3 at 6 and 10 seconds respectively. We would then use the passed time to linearly interpolate between the two positions.

```
D3DXVECTOR3 KeyFrameAPos  = ( 40 , 0 , 0 );   // Low Closest Key Frame '2'
float       KeyFrameATime = 6;

D3DXVECTOR3 KeyFrameBPos  = ( 50 , 20 , 0 );  // Higher Closest Key Frame '3'
float       KeyFrameBTime = 10;

float       PassedTime  = 8 ;       // The time out application requested

float s =  PassedTime - KeyFrameATime;       // = 2
      s /= (KeyFrameBTime - KeyFrameATime);  // = 2 / ( 10-6 ) = 0.5

Current Position  = KeyFrameAPos + ( s    *  (KeyFrameBPos - KeyFrameAPos );

            // = ( 40 , 0 , 0 )  + ( 0.5  *  ( 10 , 20 , 0 ) );
            // = ( 40 , 0 , 0 )  + ( 5 , 10 , 0 );
            // = ( 45, 10 ,0 )
```

The final position calculated would place our ball exactly half way along the green direction vector between keyframes 2 and 3 in the diagram.

So far we have looked only at translation-based animation examples, but often, animations will contain rotation and scaling sequences as well. The concepts are identical in these cases -- scaling will also use a linear interpolation approach (just like we saw with translation key frames) while rotations (stored as quaternion keyframes) will use a spherical linear interpolation (i.e. slerp) to accomplish the same objective. While these are the only keyframe types that D3DX supports in its animation controller, you could certainly use keyframes to manage other types of events in your engine. In Module III of this course series for example, we will process a set of keyframes that store color and transparency events for animating particles.

D3DXAnimationSets can store their keyframe data for a given animation in two storage styles. The first style uses separate lists for the individual transformation types (translation, rotation, or scaling). This is referred to as the SRT storage model (**S**cale**R**otate**T**ranslate). There is a list of scale vectors, a separate list of translation vectors and a list of rotation quaternions describing all the scale, translation and rotation keys assigned to that single animation respectively (the SRT for a single hierarchy frame). Every instruction in each list is assigned the time at which the instruction should be applied to the attached frame. Keyframe interpolation has to be done for each list in isolation (for times in between keyframes) to generate a final scale, position and orientation which are used to update the attached frame matrix. This is referred to as *SRT interpolation*. In the animation controller, an animation is really just a list of keyframes (usually in SRT format) and a frame name describing the name of the frame to which the keyframes are applied.

The second storage style uses a single list of 4x4 matrices that represent the combination of all three transform types. In this format, each animation is stored as a single list of matrix keys along with their assigned times. When this time is reached in the animation set's local timeline, the corresponding matrix will become the frame matrix to which it is attached. For times in between keyframes, the two matrices that most closely bound the desired time will be used to interpolate a new matrix that will be used instead.

While it may seem more convenient to store animations as a single list of matrix keys, there are times when storing matrix keyframes is less than ideal (we will discuss this later). As programmers, the choice of whether each animation's keyframes are stored as matrix keys or as separate scale, rotation and translation keys (SRT Keys) is usually out of our hands. We have to adapt our approach according to how that data is stored in the X file. The modeler or artist who created the X file may well have a choice when exporting their data to X file format as to which keyframe data format they prefer, so if you have preference for your engine, it is best to let them know in advance.

Once we know how the keyframe animation data is stored in an X file, things will start to make a lot more sense. Let us explore that concept in the next section.

10.3 X Files and Animation

Animation data is stored in an X file using three standard templates. We will look at each in turn.

The first template we will study is the **Animation** template, which manages the keyframe data for a single frame in the hierarchy. When the D3DX loading function encounters an Animation data object, it will create and add a new animation to the animation set currently being constructed. The animation itself (which is just an array of keyframes) will be populated by the keyframes specified in the Animation data object in the X file. Therefore, the Animation data object in an X file is analogous to an 'Animation' in the D3DX animation system. The X file Animation data object will contain one or more **AnimationKey** child data objects that store the actual keyframe data. There will also be a child data reference object which is the name of the frame in the hierarchy for which this keyframe data applies.

Using our automobile example from the last chapter, let us assume that our artist has applied rotation data to the wheels. In this case there would be four Animation data objects in the file -- one for each wheel frame in the hierarchy. D3DX would create an animation set with four animations. Each animation would be attached to a wheel frame.

The Animation template is an open template defined in DirectX as follows:

```
template Animation
{
      <3D82AB4F-62DA-11cf-AB39-0020AF71E433>
      [...]
}
```

The first thing an object of this type will need is the name of the frame in the hierarchy that it animates. For example, in our bouncing ball example we might give the frame that stores the ball the name 'Ball_Frame'. We would embed this name in the Animation object in the X file as follows:

```
Animation BallAnimation
{
      { Ball_Frame }
}
```

Now that the Animation object is linked to a frame in our hierarchy, we will need to add some keyframe data. If the animation has been saved in SRT format (scale, rotation, translation) then there could potentially be three AnimationKey data objects that exist as children of the Animation object.

Each AnimationKey data object will contain a list of one or more keyframes for a specific key type. For example, the first AnimationKey object may contain a list of Scale keyframes, where each keyframe describes the scale for the attached frame matrix at a particular point in the timeline. The second AnimationKey object might contain a list of Rotation keyframes where each keyframe in this list describes the orientation of the attached frame at a given point along the timeline. Finally, the third AnimationKey object might contain a list of Translation keyframes describing the position of the attached frame at key points in the animation timeline. In the SRT format we can see that all three transformation types are stored separately, although this is not necessarily a one-to-one mapping. For example, we may have 10 scale keyframes, 150 rotation keyframes, and 0 translation keyframes. In this example the Animation object would contain only two child AnimationKey objects because there is no translation data to be stored. If the animation data is stored in matrix format instead of SRT format, the Animation data object will always contain only a single child AnimationKey data object that acts as a container for the animation matrix keys.

The AnimationKey template is a standard X file template and is defined as follows:

```
template AnimationKey
{
      <10DD46A8-775B-11cf-8F52-0040333594A3>
      DWORD keyType;
      DWORD nKeys;
      array TimedFloatKeys keys[nKeys];
}
```

DWORD KeyType

The AnimationKey object can store individual transformation components or a combined matrix. This first member identifies the type of keyframe data contained in the object. The possible values are shown in the table below.

KeyType	Description
0	The AnimationKey object will contain a list of rotation keyframes. Each key frame stores a rotation quaternion in a 4D vector representing the parent relative orientation of the frame.
1	The AnimationKey object will contain a list of scale keyframes. Each keyframe will store a 3D vector describing how to scale the frame along its X, Y, and Z axes respectively.
2	The AnimationKey object will contain a list of translation vectors. Each keyframe in the list will be represented as a 3D vector describing the position of the frame relative to its parent frame.
4	The AnimationKey object will contain a list of matrix keyframes. Each keyframe will be a 4x4 matrix containing the relative scale, rotation, and translation information for the frame at the assigned time

Note: The SDK 9.0 documentation states that a value of '3' denotes the AnimationKey as containing matrices. This is incorrect and the correct value to identify an AnimationKey as a matrix keyframe container is '4' as shown in the above table.

DWORD nKeys

The second member of the AnimationKey template describes the number of keyframes in the list that follows. For example, if the KeyType member was set to 2 and the nKeys member was set to 22, this would indicate that the animation key contains 22 translation keyframes.

array TimedFloatKeys keys[nKeys]

The final member of the AnimationKey template is an array of TimedFloatKeys, which are the keyframe data. Each element in this array is a single keyframe and the array will be large enough to hold as many keyframes as are described in the nKeys member.

The TimedFloatKey type is another standard template and is defined as follows:

```
template TimedFloatKeys
{
    < F406B180-7B3B-11cf-8F52-0040333594A3 >
    DWORD time;
    FloatKeys tfkeys;
}
```

An object of this type will exist for each keyframe in the array. The first member contains the time where this keyframe can be found on the timeline. We will discuss time units (seconds, milliseconds, etc.) shortly. The FloatKeys child object is another standard template object:

```
template FloatKeys
{
   < 10DD46A9-775B-11cf-8F52-0040333594A3 >
   DWORD nValues;
   array float values[nValues];
 }
```

The FloatKeys object allows us to store arbitrary length vectors. Rotation keyframes are represented as 4 floats (a quaternion), while scaling and translation keyframes require only 3 floats. We can also store matrix keyframes as a 4x4 matrix (16 floats). The nValues member will tell us how many floats are stored in the array.

TimedFloatKeys data objects are really just wrappers around FloatKeys data objects with an additional time component.

With all of this in mind, let us examine a simple X file with two animated frames. The first frame (MyFrame1) will include rotation and translation animations, but no scaling. This means the Animation object for this frame will have two AnimationKey child objects -- one containing a list of rotation keyframes, the other a list of translation keyframes.

The second frame (MyFrame2) will use a list of matrix keyframes instead of separate scale, rotation and translation lists. As a result, the Animation object for this frame will contain a single AnimationKey child data object with a list of matrix keyframes.

We will look only at the relevant part of the X file so you can just assume that the frame hierarchy and meshes are stored elsewhere.

```
Animation   Anim1     // Animation Data for 1st animated frame
{
      { MyFrame1 }

      AnimationKey
      {
            0;    // Rotation Key Frames
            4;    // Four rotation key frames in list

            //    Time  NumFloats   Quaternion Data
            0;    4;    1.00000 ;  0.00000 ; 0.00000 ; 0.70000;;;  // Key Frame 1
            5;    4;    0.22222 ;  0.00000 ; 0.03333 ; 0.20000;;;  // Key Frame 2
            10;   4;    0.02000 ;  0.00000 ; 1.00000 ; 0.70000;;;  // Key Frame 3
            20;   4;    0.22222 ;  1.00000 ; 0.03333 ; 0.20000;;;  // Key Frame 4
      }

      AnimationKey
      {
            2;    // Translation Key Frames
            3;    // 3 Translation key frames in list

            //    Time  NumFloats   3D Translation Vectors
            0;    3;     0.00000 ; 0.00000 ; 0.70000;;; // Key Frame 1
            2;    3;     0.00000 ; 4.03333 ; 0.20000;;; // Key Frame 2
            4;    3;     0.00000 ; 1.00000 ; 3.70000;;; // Key Frame 3
      }
}

Animation Anim2    // Animation data for 2nd animated frame
{
      {MyFrame2}

       AnimationKey
       {
            4;    // This Animation Key holds Matrix Key Frame Data
            3;    // Three Key Frames

            //    Time  NumFloats   Matrix Data
            0;    16;   1.00000;    0.00000;    0.00000;    0.00000; // Key 1
                        0.00000;    1.00000;    0.00000;    0.00000;
                        0.00000;    0.00000;    1.00000;    0.00000;
                        0.00000;    0.00000;    0.00000;    1.00000;;;
            20;   16;   1.00000;    0.00000;    0.00000;    0.00000; // Key 2
                        0.00000;    1.00000;    0.00000;    0.00000;
                        0.00000;    0.00000;    1.00000;    0.00000;
                        10.00000;   0.00000;    0.00000;    1.00000;;;
            30;   16;   1.00000;    0.00000;    0.00000;    0.00000; // Key 3
                        0.00000;    1.00000;    0.00000;    0.00000;
```

```
                        0.00000;    0.00000;    1.00000;    0.00000;
                        20.00000;   30.00000;   0.00000;    1.00000;;;
      }
}
```

Notice that there are three semi-colons at the end of each keyframe. This is because, in addition to ending the float (one semi-colon), we are also closing the FloatKey object (another semi-colon) and the outer TimedFloatKey object (the third semi-colon). For more information on why these semi-colons appear, please read the section in the DX9 SDK docs entitled "Use of Commas and Semicolons".

> **Note:** Quaternions are stored inside X files with their components in WXYZ order, not the more typical XYZW format of 4D vectors.

It is important to emphasize that D3DXLoadMeshHierarchyFromX will automatically load the keyframe data (regardless of the format it is in) and internally store the data for each of its animations inside an animation set in **SRT format**.

The next template we will look at is called AnimationOptions. This is another potential child object of the Animation data object (the X file version of a 'frame animation') and is part of the standard template collection. This object will tell us whether this frame animation is a looping animation or not as well as whether interpolation between keys should be linear or spline based (splines are curves and will be discussed later in the series). The template used to describe the animation options is:

```
template AnimationOptions
{
  < E2BF56C0-840F-11cf-8F52-0040333594A3 >
    DWORD openclosed;
    DWORD positionquality;
}
```

The first DWORD indicates whether the animation is open or closed. If we set this value to '0' then the animation is described as a 'closed' animation and it will not loop. If we set this value to '1' then the animation is 'open' and should loop.
The second member describes whether linear interpolation or spline-based interpolation should be performed between keyframes. A value of 0 requests cheaper but lower quality linear position interpolations, while a value of one requests smoother but more expensive spline-based position interpolations.

> **Note:** Based on some internal testing results, it would appear that for the moment at least, this template type is not properly recognized by the D3DX animation controller (DX 9.0b). Please keep this in mind if you choose to use this template in your own X files.

The next animation related X file template that we will examine is the AnimationSet:

```
template AnimationSet
{
      <3D82AB50-62DA-11cf-AB39-0020AF71E433>
      [Animation]
}
```

The AnimationSet data object is a container for Animation objects (note that it is a restricted template). All Animation objects will be contained within a parent AnimationSet. In the previous chapter on hierarchies, you will remember that we actually wrote our own CAnimationSet class that was basically a container for CAnimation objects. The relationship between the X file AnimationSet data object and the X file Animation data object is analogous to the relationship between our proprietary CAnimationSet class and the CAnimation class. However, as mentioned, the data for a single frame animation is not stored in its own objects but stored directly inside the animation set to which it is assigned (likely in a large array of animations). The animation set is also more than just a container; it also provides the logic to interpolate between the SRT keyframes stored in each of its animations to generate the frame matrix for each animation at a specific periodic position in its timeline. The animation set also has its own local timeline, although that is managed at the animation controller level by the animation mixer (more on this later).

We will see later in this chapter that we can actually have multiple AnimationSets assigned to the same hierarchy. This is especially useful when animating a character hierarchy, where one animation set would contain all of the Animations (i.e., frame animations) to make the character walk while a second animation set would contain the Animations to make the character fire a weapon. Both animation sets would reference the same hierarchy, and indeed the Animations in two different animation sets will often animate the same hierarchy frame. Since the D3DX animation controller allows us to mix multiple animation sets together, we can even simultaneously blend the walking animation with the weapon firing animation so that a character can fire while on the move. We will return to explore this concept in great detail later in the lesson.

Next we see how animation data might look in a stripped-down X file using the animation set concept just discussed.

```
AnimationSet Walk // Walking Animation
{
      Animation   // Animation data for 1st animated frame
      {
            {MyFrame1}
            AnimationKey
            {
                  4;    // This Animation Key holds Matrix Key Frame Data
                  3;    // Three Key Frames
                  Key-Frame Matrix Data Goes here
            }
      }

      Animation   // Animation data for 2nd animated frame
      {
            {MyFrame2}
            AnimationKey
            {
                  4;    // This Animation Key holds Matrix Key Frame Data
                  3;    // Three Key Frames
                  Key-Frame Matrix Data Goes here
            }
      }
}  // end Walk animation set
```

```
AnimationSet Shoot     // Shooting animation
{
      Animation   // Animation data for 1st  animated frame
      {
            {MyFrame1}
             AnimationKey
             {
                  4;    // This Animation Key holds Matrix Key Frame Data
                  3;    // Three Key Frames
                  Key-Frame Matrix Data Goes here
             }
      }

      Animation   // Animation data for 3rd animated frame
      {
            {MyFrame3}
             AnimationKey
             {
                  4;    // This Animation Key holds Matrix Key Frame Data
                  3;    // Three Key Frames
                  Key-Frame Matrix Data Goes here
             }
      }
}  // end 'Shoot' animation set
```

Note that you do not have to give an animation set a name since they can be referenced by index once they have been loaded by D3DX. However, giving them a name is always a good idea if given the choice in your X file exporter. The D3DXLoadMeshHierarchyFromX would load the sample X file and would create two animation sets that would then be registered with the animation controller. Each animation set in this example would have two animations.

In the walk animation set above, Frame1 and Frame2 of the hierarchy are manipulated. For the shoot animation, Frame1 and Frame3 are manipulated. If these animation sets were blended together, then Frame1, Frame2, and Frame3 would be animated simultaneously. Frame1 would be influenced by both animation sets because there would be an animation in both animation sets attached to it and updating its matrix.

Unfortunately, at least at the time of this writing, many X file exporters are able to save only a single animation set in the X file. This is true regardless of how many logical types of animations are created for the scene (walk, run, jump, etc.). To be clear, the frames for these logical animations are all saved; they just happen to be stored in the same animation set in many cases. While you can usually configure the modeller to export only a specific range of keyframes, often the entire hierarchy will need to be saved as well. However even if you can avoid this, or code around it, or simply do not care about the extra space, it is still much easier to manage your animation data if it is not all stored in a single set.

On the bright side, in Lab Project 10.2 we will create a small tool to circumvent this problem. Our GUI tool will allow the user to load an X file that contains all of its animations in a single animation set and then use the D3DX animation interfaces to break those animations into separate named animation sets before resaving the file. Apart from being a very handy tool, it will also get us acquainted with using the D3DX animation interfaces to build an ID3DXAnimationController and ID3DXAnimationSet objects

from scratch. This is something you will definitely want to learn how to do if your animation data is stored in a file format other than X.

10.3.1 Timing and X Files

As it turns out, the timing values stored within the keyframes of an X file are not specified in seconds; time is instead specified using *ticks*. By itself, this timestamp format is fairly useless, for it is largely arbitrary, and depends on factors such as how fast we want the animation to run and how many of these ticks the exporting application (the 3D modelling program) was processing per second. For D3DX to process these keyframe timestamps in a meaningful way and convert them into seconds, there is an additional X file template, called `AnimTicksPerSecond`, which can be placed anywhere in the file. It contains a single DWORD which tells us (and D3DXLoadMeshHierarchyFromX) how many of these ticks equal one second in application time. The D3DX loading function uses this data to convert keyframe timestamp values stored in the X file into seconds for storage within its interpolator objects.

```
template AnimTicksPerSecond
{
      DWORD NumberOfTicks;
}
```

As it turns out, this data is omitted from most X files. If it is not included, D3DX assumes a default of 4800 ticks per second. This is the default also assumed by most 3D modelling applications which export to the X file format. This means that if the last keyframe in any given animation has a timestamp of 67200, the animation will have a length of 14 seconds. Although this may seem to be an unnecessary middle level of interpretation introduced at load time, it does allow for the easy alteration of an animation's speed and length by simply changing the number of ticks per second value stored in the X file. Perhaps we might find that the animation data exported from a certain application runs too fast and we would like to slow it down. Without this Ticks Per Second concept, we would have to manually adjust the timestamps of every keyframe in the X file (a laborious process indeed).

There are a few traditional `AnimTicksPerSecond` values that many developers/modellers use beyond the default of 4800. Values like 30, 60, 24, and 25 are often used as well. These values generally correspond to 'frames per second' standards that are used across the industry. If we were to use the 30 ticks per second case as an example, this would mean that for a keyframe to be triggered ten seconds into the animation, its timestamp should be 300. Again, this is all handled on our behalf by the D3DX animation controller and its animation sets, so it may not seem relevant. However, it certainly helps to understand how the timing is interpreted when building animations manually, or when writing your own exporter.

Before moving on to the next section, let us quickly sum up what we know so far. First, we now have a good understanding of how animation data is represented in an X file. We know that there will usually be a single animation set, and it will contain one or more animation data objects. Each animation data object is connected to a single frame (by reference) in the frame hierarchy (contained elsewhere in the X file) and stores the keyframe data for that frame, essentially describing that single animation's timeline. We know that inside the animation object, the keyframe data can be stored in SRT format, where there

will be three separate keyframe lists (three AnimationKey objects) for scale, rotation, and translation data. Alternatively, the data can be stored as a single AnimationKey using a keyframe list of matrices storing the combined scale, rotation, and translation information. We also know that regardless of how the data is stored in the X file, once loaded, the animation data will be stored inside the animation set as SRT lists for each animation defined. We learned that the timestamp of each keyframe in an animation is defined in the X file using an arbitrary unit of measurement called a tick, which is given meaning by the exporter by either inserting an AnimTicksPerSecond data object somewhere in the file or by using the D3DX default (4800 ticks/sec). Finally, we know that more often than not, we will let D3DX do all of the hard work for us. We will simply call ID3DXAnimationController::AdvanceTime to generate/update the relative matrices for each frame to correspond to the elapsed time for the currently active animation(s).

10.4 The D3DX Animation Interfaces (Overview)

Now that we know how our animation data is stored at the file level, let us examine how it is arranged in memory once D3DX has loaded it using D3DXLoadMeshHierarchyFromX. As it turns out, there is not much difference between the two cases.

When D3DXLoadMeshHierarchyFromX loads in animation data, that data is stored and managed by a D3DX animation subsystem consisting of two different COM object types. This exposes two main animation interfaces to our application (there are others used for intermediate tasks). We will look at each of them very briefly here first, and then in more detail as the chapter progresses.

10.4.1. The Animation Controller

The top level of the animation subsystem is the ID3DXAnimationController interface. An interface to this object is returned to our application by the D3DXLoadMeshHierarchyFromX function. It is this interface that our application will work with most frequently. We can think of it as being an animation set manager -- when animation sets are loaded from the X file, each animation set is registered with the newly created animation controller and an interface to this controller is returned to the application.

The animation controller also provides a very powerful *animation mixer* that can be used to assign animation sets to different ‘tracks’ and blend them together when the animations are played. The animation mixer can be thought of as being somewhat like a multiple-track audio mixing desk, where we would feed in individual pieces of audio data (drums, guitar, vocals, etc.) on different ‘tracks’ or ‘channels’ and blend them together based on the properties that we have set for each track (ex. volume, bass, treble, etc.). What comes out of the mixing desk output jack is all of the audio data, from all of the tracks, combined together into a final product that can be saved to CD and played back.

If we needed to play only a single animation set and did not require blending, the animations sets would all still be loaded and registered with the animation controller by the D3DXLoadMeshHierarchyFromX function; we would just assign the animation set we wish to play to track 1 and play it back by itself.

Note: Even if an X file contains multiple animation sets, by default D3DXLoadMeshHierarchyFromX will assign the only last animation set defined in the file to track 1 of the mixer. No other mixer tracks or animation sets will be set up for you, so if you wish to play them, you will have to assign these animation sets to the mixer tracks manually. This is why you will not see us setting up the animation mixer or doing any kind of animation set initialisation in Lab Project 10.1. The X file only contains a single animation set and it is assigned to track 1 by default. When the loading function returns, we are ready to start playing the animation set immediately.

10.4.2. The Animation Set

The ID3DXAnimationSet interface and its underlying COM object mirrors the X file AnimationSet template almost exactly. This is the base class interface and it is actually the ID3DXKeyframedAnimationSet that we will be using most of the time. There is also an ID3DXCompressedAnimationSet which is an animation set containing compressed keyframe data. Once keyframe animation data has been loaded by D3DX into an ID3DXKeyframedAnimationSet, we can use the methods of this interface to create a compressed animation set if we wish.

The base interface (ID3DXAnimationSet) is pure abstract, thus allowing you to derive your own animation set classes which interpolate between the keyframes of each of its animations in a customized manner. As long as the derived class exposes the functions of the base interface, the animation controller will happily work with it to generate the frame matrices without caring how that matrix data was generated.

For each AnimationSet data object contained in the X file, D3DXLoadMeshHierarchyFromX will create a new D3DXKeyframedAnimationSet object and register it with the animation controller. On function return, we can use the methods of the ID3DXAnimationController interface to assign these animation sets to tracks on the mixer so that they are ready for playback. Each animation set will contain the SRT data of each animation data object stored in the file. It will also contain a mapping describing which keyframe animation data sets are assigned to which frame(s) in the hierarchy. This is a simple name match. In other words, the name of an animation within an animation set matches the name of its assigned frame in the hierarchy.

Recall that an X file animation object is essentially a container of AnimationKey data objects for a single frame in the hierarchy. This same relationship is maintained in D3DX. In this case however, each Animation Set will contain the array of SRT keyframes along with a frame name. It is the SRT keyframe set for a single frame in the hierarchy which is referred to as being a ***frame animation***, or an ‘Animation’ for short. The contents of each Animation inside of an animation set are equivalent to the X file AnimationKey data object contents. For each frame that is animated in the hierarchy, a matching animation will be created and attached to that frame and managed by D3DX via its parent animation set. The Animation stores all the keyframe data for that frame for the animation set to which it is assigned.

For example, if an X file contains an animation set that animates five frames in the hierarchy, a D3DXKeyframedAnimationSet object will be created and registered with the animation controller, and five Animations will be created and stored inside it. They will contain the individual keyframes for each

of the animated hierarchy frames. These animations are owned by the animation set and the animation set will be owned by the animation controller.

We will hardly ever need to work with the ID3DXAnimationSet interface or the ID3DXKeyframedAnimationSet interface when playing back an animation. Often, our application's only exposure to these interfaces will occur when we wish to use the animation mixer to blend animation sets, or assign animation sets to mixer tracks. For example, ID3DXAnimationController has a function called SetTrackAnimationSet which takes two parameters: a track number and an ID3DXAnimationSet interface. Calling this method will assign the passed animation set to the specified track on the animation controller's mixer. When using the animation controller in this simple way, we will often have no need to actually call any of the animation set's methods. The methods of the animation set are usually called by the animation controller to generate the SRT data for each animated frame in the hierarchy (in response to a call to AdvanceTime from the application).

Keyframe animation sets are much more than simple animation containers. In fact, when our application calls ID3DXAnimationController::AdvanceTime, the animation sets manage much of the total animation process. They select the appropriate keyframes for a given moment in time for each animation and then use them to generate dynamic animation data that transforms the frames to which the animations are attached.

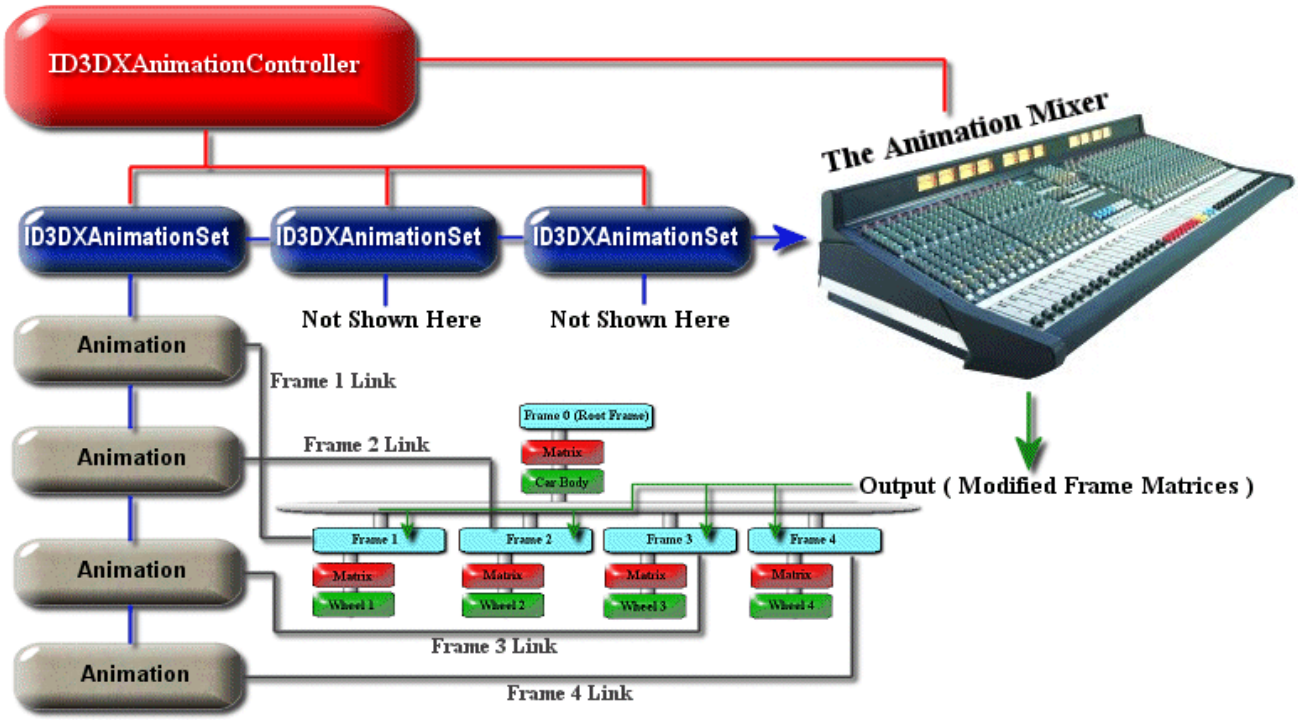
When ID3DXAnimationController::AdvanceTime is called, the animation controller will begin by looping through each of its currently active AnimationSets (an *active* set is one that has been assigned to a track on the animation mixer). For each animation set, it will use the ID3DXAnimatonSet interface to fetch each of its underlying animations. For each animation, the controller will call the ID3DXAnimationSet::GetSRT function. GetSRT will use the current animation local time (i.e., periodic position) of the animation set to search for its two bounding keyframes and then interpolate a new value somewhere in between. The output is a new position, scale and rotation for the frame to which it is attached. The animation controller will do this for each animation contained in each animation set. At the end of this process, the animation controller will contain the new matrix information for each animated frame in the hierarchy. These may be placed directly into the frame matrices or, depending on how the mixer is configured, this matrix data might be blended together to create the final matrix data for the assigned frame in the hierarchy. So, in the case of the ID3DXKeyframedAnimationSet interface, the animation set is responsible for performing the actual keyframe interpolation and handing the results back up to the animation controller for mixing.

The animation controller will typically cache the new frame position, scale, and rotation information generated by the ID3DXAnimationSet::GetSRT function call (for each animation) until all active animation sets have been processed. Then the new frame data for each animation set will be scaled by the weight assigned to its track. If two animation sets have keyframe animations defined for the same frame in the hierarchy, then the results of each animation will be blended together taking the track weights into account. The results are then stored in the parent-relative frame matrix for the appropriate owner frame in the hierarchy.

> **Note:** We can also use the ID3DXKeyFrameAnimationSet interface to directly work with the keyframe data. This can be useful if you are designing your own animation tool and need the ability to create, edit, or destroy keyframe data values. For example, we can use the ID3DXKeyframedAnimationSet::RegisterAnimationSRTKeys method to place our own keyframes directly

into an animation set that we manually create. This allows us to register animations manually with the animation set. This would be useful if you wish to use the D3DX animation system but intend to load the animation data from a custom file format. You could programmatically build the animation controller, create and register animation sets with the controller, and then add you own animations to the animation set. The keyframe data for each animation you add could be the data imported from your custom file format.

Fig 9.7 depicts an animation controller that owns three animation sets. You will notice that we are reusing our automobile hierarchy once again and that for the first animation set, there is an animation for each wheel frame (the grey lines show the frame that each animation is connected to). When we call ID3DXAnimationController::AdvanceTime, the clocks for each track on the mixer are incremented by this value. Once the timers for each track have been updated, the current time of the track is mapped to the local time of the animation set assigned to it (to account for looping, etc). Each animation set uses this time to calculate a new position, rotation, and scale for each animation by interpolating its stored keyframes (via its GetSRT function). It passes the results back up to the animation controller where they are fed into the animation mixer. The result of the mixing process is a new frame matrix generated using any or all of the animation sets results. This data is stored in the frame matrices so that the next time we update and render the hierarchy, the matrices have been adjusted and the children (meshes, other frames) appear in their new position or orientation.



The D3DX Animation System

Figure 9.7

So we actually have several timers at work here. The global time of the animation controller is the time that is updated every time our application calls the ID3DXAnimationController::AdvanceTime method.

We pass in an elapsed time value that is used to update the controller's timer variable. This same elapsed time is also added to the timers of each active track on the mixer (a track that has an animation set assigned to it). We can think of these as being global times for a given track. By keeping a global time for each track, we have the ability to advance the time of a single track without affecting any of the others. If we only had a single global time shared by all tracks, altering the position of the global time would cause all tracks to be updated.

Finally, while we might think that the track time is the time used by the animation set to fetch its keyframes, this is not the case. The track time is still a global time value that has no consideration for the duration of the animation or whether it is used to play in a looping or ping-pong fashion. Therefore, once the timer of a given track has been updated, the controller will use the ID3DXAnimationSet::GetPeriodicPosition method to map the track time into a local time for the animation. This local time is referred to as the *periodic position* of the animation and it is this periodic position that is passed into the ID3DXAnimationSet::GetSRT method by the controller to fetch the SRT data for a given animation. We will discuss the periodic position of an animation set and the mapping of track time into local animation time more a bit later in the chapter. Just know for now, that this is all handled on our behalf by the animation controller.

10.5 SRT vs. Matrix Keyframes

Earlier we discussed the fact that X file keyframe data can be stored in one of two ways: SRT transforms can be maintained in separate lists or they can be combined into a single list of 4x4 matrices. Regardless of the file keyframe format, the ID3DXKeyframedAnimationSet always stores keyframe data in SRT format. Internally, the D3DXKeyframedAnimationSet maintains a list of private arrays for **each** animation, similar to what we see below:

```
LPCSTR                  AnimationName;
LPD3DXKEY_VECTOR3       pTranslationKeys;
LPD3DXKEY_QUATERNION    pRotationKeys;
LPD3DXKEY_VECTOR3       pScaleKeys;
ULONG                   NumTranslationKeys;
ULONG                   NumRotationKeys;
ULONG                   NumScaleKeys;
```

Each animation maintains three separate lists of keyframe data and stores an animation name so that it knows which frame in the hierarchy these keyframe lists affect. The name of the animation will be the name of the frame to which it is attached. Translation and scaling keyframes are both stored in D3DXKEY_VECTOR3 arrays. This structure contains a time value and a 3D vector:

```
typedef struct _D3DKEY_VECTOR3
{
      FLOAT       Time;
      D3DXVECTOR3 Value;
} D3DKEY_VECTOR3, *LPD3DXKEY_VECTOR3;
```

When used to store translation keyframes, the 3D vector contains the position of the frame relative to its parent frame. When this structure is used to store scale keyframes, the 3D vector stores scaling amounts for the frame matrix along the X, Y, and Z axes.

Rotation keyframes are stored as an array of D3DXKEY_QUATERNION structures. This structure includes a timestamp in addition to a standard quaternion.

```
typedef struct _D3DKEY_QUATERNION
{
      FLOAT             Time;
      D3DXQUATERNION    Value;

} D3DXKEY_QUATERNION, *LPD3DXKEY_QUATERNION;
```

The D3DXQUATERNION structure is a 4 float structure (X,Y,Z,W) that will store the frame orientation relative to the parent. D3DXLoadMeshHierarchyFromX takes care of converting quaternions stored in the X file in WXYZ format to XYZW format before storing them in the interpolator quaternion array.

> **Note:** 3D Studio Max™, Milkshape™ and other animation editors also store their keyframe data in SRT format, not as a single array of matrices. If you are familiar with 3D Studio Max™ you will notice that on the timeline bar, when a keyframe is registered, there are three different colored blocks which denote the type of 'event' is tracked at that time (scale, rotation, or translation). If only a translation occurred, there will only be one colored block there. The animation information layout in the .3DS file format also stores its keyframe animation in an almost identical manner to that discussed above.

It is worth noting that even if your original keyframe data was stored as a matrix list, those matrices would be split into SRT components before they are stored in the animation's SRT arrays by D3DX. Why do this? After all, matrix keyframes seem like a nice idea since we only have to maintain a single array of keyframe data.

As it turns out, there are actually a few good reasons why SRT lists are preferred over matrices. One of the more obvious is memory footprint. There are 16 floats in a matrix keyframe and a maximum of 10 in an SRT keyframe (and perhaps fewer depending on which components are active).

But more importantly, accurate interpolation between matrices is essentially not possible under certain circumstances. This is a direct result of the fact that scaling data and rotation data become intermixed, and are ultimately indistinguishable in the upper 3x3 portion of the matrix.

Consider the following example animation data. In our editing package, we create two cubes side by side. One of these we wish to **scale** by a factor of (`X:-1, Y:1, Z:-1`) over the course of the animation ('the animation' being a simple two keyframe animation) and the other is **rotated** 180 degrees around the Y axis. So in SRT keyframe format, we would end up with two keyframes for each cube:

```
Animation {

  { Cube01 }

  AnimationKey {
```

```
    1;       // 1 = Scale
    2;       // 2 Key frames
    0;     3;  1.000000, 1.000000,  1.000000;;,
    67200; 3; -1.000000, 1.000000, -1.000000;;;
  }
}

Animation {

  { Cube02 }

  AnimationKey {
    0;       // 0 = Rotation
    2;       // 2 Key frames
    0;     4; 0.000000, 0.000000, 0.000000, 1.000000;;.
    67200; 4; 0.000000, 1.000000, 0.000000, 0.000000;;;
  }
}
```

This works fine and we can see that when we run the animation, each cube acts as we would expect it to when the animation is played out. Interpolation of each cube's animation data will produce expected results for any time within the timeline. In the case of the first cube, two scale keyframes are being interpolated, causing the cube to slowly invert itself over the duration of the timeline. In the case of the second cube, two rotation keys will be used for interpolation, causing the cube to turn 180 degrees over the duration of the timeline. If we were to switch over to matrix keys however, let us take a look at the matrix keyframe values that would now be stored in the X file:

```
Animation {
  { Cube01 }
  AnimationKey {
    4;       // 4 = Matrix Key
    2;       // 2 Key frames
    0;     16; {MatrixA};;,
    67200; 16; {MatrixB};;;
  }
}

Animation {
  { Cube02 }
  AnimationKey {
    4;       // 4 = Matrix Key
    2;       // 2 Key frames
    0;     16; {MatrixA};;,
    67200; 16; {MatrixC};;;
  }
}
```

The matrices listed above are just placeholders, so let us have a look at the actual values. MatrixA and MatrixB describe the two matrix keyframes of the first frame (which is to be scaled) and MatrixA and MatrixC describe the two matrix keyframes for the second animated frame (which is to be rotated).

```
{MatrixA}     1.000000, 0.000000, 0.000000, 0.000000,
              0.000000, 1.000000, 0.000000, 0.000000,
              0.000000, 0.000000, 1.000000, 0.000000,
              0.000000, 0.000000, 0.000000, 1.000000

{MatrixB}    -1.000000, 0.000000,  0.000000, 0.000000,
              0.000000, 1.000000,  0.000000, 0.000000,
              0.000000, 0.000000, -1.000000, 0.000000,
              0.000000, 0.000000,  0.000000, 1.000000

{MatrixC}    -1.000000, 0.000000,  0.000000, 0.000000,
              0.000000, 1.000000,  0.000000, 0.000000,
              0.000000, 0.000000, -1.000000, 0.000000,
              0.000000, 0.000000,  0.000000, 1.000000
```

We can see that the first cube (the cube we are attempting to scale by -1 on both the X and Z axes) uses both MatrixA and MatrixB as its keyframes. MatrixA is an identity matrix (for both frames) so the cube meshes attached to those frames will start off aligned with the world X, Y, and Z axes facing down +Z. We can see that MatrixB is a scaling matrix. At 67200 ticks, the first cube will scale itself such that it will then be inverted and facing down -Z. So far, this is all essentially fine.

However, have a close look at MatrixC which is used for the second cube's final keyframe. This defines our 180 degree Y rotation keyframe. You will notice that MatrixB and MatrixC contain the exact same values. In fact, a two axis negative scaling matrix is identical to a 180 degree rotation matrix on the third (unused) axis. At first you might think that this is not a problem. After all, the final result you want in both cases is identical, and thus the matrices should be identical too, right? Certainly, in both cases, at timestamp 67200, both cubes would be facing down the -Z axis and would look identical.

The problem occurs when the animation set for which the animation is defined is trying to interpolate between the two keyframes (A/ B and A/C) to generate the frame matrix for a time that falls between their timestamps. How would the interpolation process know whether it should gradually scale the object from one keyframe to the next or whether it should rotate it between keyframes? Take a look at the following two matrices which show how the matrices in between the keyframes in our example might look, depending on whether the two keyframes are supposed to represent a scale or a rotation:

The first matrix is 50% through a 180 degree rotation about the Y axis. This is the frame matrix we would expect the interpolation process to generate for our rotating cube frame exactly halfway through its timeline:

```
0.000000, 0.000000, -1.000000, 0.000000,
0.000000, 1.000000,  0.000000, 0.000000,
1.000000, 0.000000,  0.000000, 0.000000,
0.000000, 0.000000,  0.000000, 1.000000
```

The second matrix is 50% through a -1.0 scaling along the X and Z axes. This is the frame matrix we would expect the interpolation process to generate for our scaling cube frame exactly halfway through its timeline.

```
0.000000, 0.000000, 0.000000, 0.000000,
0.000000, 1.000000, 0.000000, 0.000000,
0.000000, 0.000000, 0.000000, 0.000000,
0.000000, 0.000000, 0.000000, 1.000000
```

These two matrices are clearly quite different, even though carrying on interpolating all the way to the end of the animation (100%) would result in identical matrices, as we have seen. This shows that while the keyframe matrices of both a 180 degree Y axis rotation and a –1.0 XZ axis scale are identical, the matrices generated by the interpolation at times that fall between the two keyframes can be very different.

So how exactly does D3DX address this problem when converting matrix keyframes stored in an X file to separate SRT lists for the animation stored inside the animation set? How does it look at two keyframe matrices (such as those in our example) and determine whether these two keyframes represent a scale or a rotation? The answer is: it does not know. There is no way to distinguish between our scale and rotation examples because in both cases the two keyframes for each animation are identical, even though we would desire different intermediate matrices during animation.

At load time, D3DX will extract what rotation, translation, and scale data it can from the matrix keyframes and store them in SRT format and hope they are correct. It will simply assume, in a case like we see above, that rotation is the correct animation and store the data in the animation's rotation keyframe list. This is generally fairly safe to assume as rotation animations are typically more common than scaling animations.

This matrix keyframe ambiguity is precisely why most modeling applications export keyframes in SRT format. Matrix keys are not completely useless however. Certain animations that require transforms that cannot easily be represented with the SRT approach (skew matrices, projection matrices, etc.) can benefit from a matrix keyframe list. In fact, this is why keyframe matrices were added to DirectX 8.0 (they did not exist in DirectX prior to that point).

There is a way around some of these matrix keyframe problems. If we were to use three key-frames for these cases instead of two, inserting our '50%' matrices outlined above into the animation, we would end up with a more accurate representation of the desired animation type. It would still have significant issues, but the more keys we add, the more accurate the animation will become. There are other cases, other than the one outlined above, where the use of matrix keys can cause problems, but generally, by sampling more keyframes during the export of the animation data, these problems can be reduced.

The bottom line is, it is probably best to favor SRT format if you have a choice when exporting your X files. Only choose a matrix format if you are sure that they will not present a problem.

10.6 The D3DX Animation Interfaces

In this next section we will examine the D3DX animation interfaces that our application will need to work with in order to animate frame hierarchies. Rather than starting at the top level of the system and covering the ID3DXAnimationController interface first (which is often the only interface our application will need to work with extensively), we will take the reverse approach and study the ID3DXKeyframedAnimationSet first. This will give us a better idea about how the animation controller uses the interface of its registered animation sets to generate the new matrices for our frame hierarchy. This discussion is not superfluous. If we understand how the system works internally, we will be better informed to use it wisely.

When we load an X file that contains keyframe animation data, one or more of these animation sets will be created for us by D3DXLoadMeshHierarchyFromX and will be registered with the animation controller. Each animation set contains a collection of one or more animations. Each animation is really just a named set of keyframes that apply to a frame in the hierarchy with a matching name when the animation set is played.

Our application will rarely need to work with the methods of the ID3DXKeyframedAnimationSet interface directly. Often, our only exposure comes when using the interface to bind animation to a track on the animation controller's internal mixer. For example, we will frequently use the ID3DXAnimationContoller interface to return one of its registered animation sets, by name or index. We will get back an ID3DXAnimationSet interface (the base class interface) which we can pass into ID3DXAnimationController::SetTrackAnimationSet to bind that animation set to a track on the mixer. Regardless of how many animation sets are registered with the animation controller, only animation sets assigned to active tracks on the animation mixer will influence the hierarchy on the next call to ID3DXAnimationController::AdvanceTime.

10.6.1 The ID3DXKeyframedAnimationSet Interface

In DirectX terminology, an *animation* is a set of keyframe lists for a single frame in the hierarchy. The *animation set* stores and manages a set of animations. Animation sets are in turn managed by the *animation controller*, which itself is the top level interface in the D3DX animation system. You may recall that this was the case based on our earlier discussions about the way the data was stored inside the X file. In our X file we saw that an AnimationSet data object contained a number of Animation data objects. These Animation data objects are the file-bound cousins of an animation in an animation set, containing keyframe data for a given frame in the hierarchy.

> **Note:** Many currently available X file exporters for 3D Studio Max™ and Milkshape™ save their data into a single animation set. This means that, without modification, these files will cause D3DXLoadMeshHierarchyFromX to create an animation controller with a single animation set containing all of the keyframe interpolators in the file. In Lab Project 10.2 we will develop a tool that allows us to modify X files so that they can contain multiple animation sets.

The ID3DXKeyframedAnimationSet mirrors the AnimationSet data object in the X file -- storing one or more animations. When D3DX loads an X file that contains animation data, it will automatically create all of the animations and incorporate them into their appropriate animation sets.

ID3DXAnimationSet exposes many methods, most of which we are not likely to need very often. This is because, when playing back animations, the animation controller will interact with the animation sets (and their underlying animations) on our behalf. All we have to do is call the controller's AdvanceTime function and all of this happens automatically. However, the interface also exposes methods to retrieve the underlying keyframe data for each of its animations so that one can directly examine the keyframes assigned to a given frame in the hierarchy. While this is not something you are likely to do on a frequent basis, it might come in handy during debugging. Studying this interface will also teach us how the entire D3DX animation system works at a low level and will provide us the knowledge to build and populate our own animation sets programmatically if we wish.

The collection of animations stored in an animation set defines the purpose of the set. Sometimes a single animation set will animate every frame in the hierarchy and under other circumstances only select frames may be affected. The latter is useful when we only wish to update portions of an articulated structure. A character is a good example; we could separate the character into logical areas (legs, torso, arms, head, etc.) and apply particular animation sets to individual parts. Consider the following animation set types for an example game avatar:

Animation Set: 1
Name: Walk
Description: This animation set contains animations which animate the leg frames of the character hierarchy to emulate the act of walking. No other animations are included for any other portion of the character.

Animation Set: 2
Name: Run
Description: Identical to the Walk animation set in that it only contains animations which animate the legs of the character. In this case, the legs are animated to give the impression that the character is running.

AnimationSet: 3
Name: Attack
Description: This animation set contains animations which animate the characters arms and torso to swing his sword.

While many more possible animation sets can exist, and some of these may even be too restrictive, the key point here is the isolation of important areas in the hierarchy. The more we can do this, the more effective will be our use of the animation mixer. As we will see later, we will be able to mix multiple animation sets together on the fly, which means that we can have our character run and attack simultaneously, without the artist having to create a specific 'Run+Attack' animation set.

You are reminded that each animation owned by an animation set may have its own unique animation length and that each animation runs concurrently. If you would prefer that all animations in a given set

finish execution simultaneously, you can do this quickly in your editing package by adding a final keyframe (at the same point in the timeline) for all frames that are animated. Remember that the duration of the animation set is the length of its longest animation (i.e., it is determined by the keyframe with the largest time stamp). If the animation set is configured to play continuously, then the animation will only loop (or ping-pong) around once this keyframe has been executed. If other animations in the set have reached their final keyframe some time before, they will remain inert until the animation set timeline restarts. The length of an animation set is referred to by DirectX as its *period* and the current position within the local timeline of an animation set is referred to as its *periodic position*.

This ID3DXKeyframedAnimationSet interface is derived from the ID3DXAnimationSet interface, which is essentially an abstract base class derived from `IUnknown`. ID3DXAnimationSet does not implement any functionality itself, but serves as the base for `ID3DXKeyframedAnimationSet` and `ID3DXCompressedAnimationSet`. It can also be inherited in order to implement your own custom SRT interpolation system if desired. Generically speaking, an animation set must generate the scale, rotation, and translation keyframe values for one of its animations based on a specified time value passed into its GetSRT member function. It returns that information via the base class functions to the animation controller.

The animation controller is only concerned with the methods exposed in the base interface, and as such, any interfaces must implement all of these methods. The ID3DXKeyframedAnimationSet interface would be useless to the controller if it did not implement the GetSRT method of the base interface. However, the ID3DXKeyframedAnimationSet interface exposes many other functions that our application can use to examine or set the keyframe data. When D3DXLoadMeshHierarchyFromX loads in animation data of the type discussed above, it will automatically create keyframed animation sets for us and register them with the returned controller. The exception to the rule is if the animation is stored in the X file in compressed format. In this case, a compressed animation set will be created instead. The controller works with both interfaces in exactly the same way, since they both expose the base class functionality that the controller desires. It is currently very rare to find X files that contain compressed animation data, so we will usually be dealing with the ID3DXKeyframedAnimationSet interface.

The base animation set interface, ID3DXAnimationSet, exposes eight methods. Each is implemented by the ID3DXKeyframedAnimationSet. While the animation controller will work with these on our behalf most of the time, if you intend to plug your own animation set interpolation system into the D3DX animation system, your derived class will need to implement these functions. This means knowing what the animation controller will expect in return when it calls them.

Base Methods

```
LPCSTR GetName(VOID);
```

This function takes no parameters and it returns a string containing the name of the AnimationSet. D3DXLoadMeshHierarchyFromX will create D3DXAnimationSet object(s) and properly assign them names based on the matching AnimationSet data object name in the X file.

The following example is the beginning of an AnimationSet data object in an X file which has been assigned the name 'Climb'. The D3DXAnimationSet for this object will be assigned the same name, and calling ID3DXAnimationSet::GetName would thus return the string 'Climb'.

```
AnimationSet Climb
{
      child 'Animation' objects defined here
}
```

You may recall from our earlier examination of X files that we are not required to assign names to data objects. Therefore it is possible that D3DX will load animation sets that have not been explicitly named in the X file. In this case, the corresponding D3DXAnimationSet will have an empty name string. If you have the choice (in your editing package or X file exporter) to assign animation sets meaningful names, it is helpful to do so. It allows you to reference the different animation sets through the animation controller using the actual name of the animation set (such as 'Walk' or 'Climb') instead of relying on a numeric index.

`DOUBLE GetPeriod (VOID);`

This function returns the total amount of time (in seconds) of the longest running animation in the animation set. The duration of an animation set is referred to as its period. If animation 1 runs for 14 seconds, but animation 2 runs for 18 seconds, this function will return 18. Indeed the first animation will have finished 4 seconds previously during playback. Only when the period value is reached in an animation set's local timeline, will its timeline be reset back to zero and both animations would being animating again (assuming the animation set is configured to loop or ping-pong). The period of the animation set is equal to the scale, rotate, or translation key with the highest timestamp among all animations in the set. If we have a scale key, a rotation key and a translation key with timestamps of 50 ticks, 150 ticks and 20 ticks, the period of the animation set would be 150/TicksPerSecond.

`DOUBLE GetPeriodicPosition ( DOUBLE Position );`

This function is used by the animation controller to get the current position within an animation set's local timeline, given a global track time (as discussed previously). Every time we pass an elapsed time value into the ID3DXAnimationController::AdvanceTime function, the animation controller adds this input time to the timer of each mixer track. Thus, the mixer track timers continue to increment with each call to AdvanceTime. When the animation controller needs to fetch the SRT data for an animation from a given animation set, it will use this method to map the track time of the animation controller into the animation set's local timeline. This is because the track time is always incremented when we call AdvanceTime and therefore it may be well outside the period of the animation set.

If the animation set is set to loop or ping-pong, then the animation set must not simply stop when the track time exceeds its period. Rather, the track time should be mapped into a local time (a periodic position) that is within the range of its keyframe data. It is this time that should be returned to the controller so that it can be used to fetch SRT data. Therefore, studying the above function, we can see that the Position parameter would contain a track time that would be passed in by the controller. This

track time needs mapping into an animation set's local time. Again, this local time is referred to as the periodic position.

This function essentially provides a mechanism for the animation set to use the track position to map into its own time (or not) however it sees fit. As the controller has no idea what data is contained in the animation set or whether or not it is set to loop, this function allows the animation set to control that process without involving the controller in the specifics. All the controller cares about is getting back a time value that it can use to fetch SRT values. How this time value is generated from the track position is entirely up to the animation set. For a non-looping animation, this function might simply test to see if the track time is larger than its highest keyframe timestamp. If so, it can pass back the timestamp of the last keyframe in the animation. This would mean that once the track time had exceeded the period of the animation set, it would appear to stop or freeze in its final position, because in every call to AdvanceTime thereafter, the period of the animation set would be returned. This would always generate the same SRT data, with the animation set in its final position.

How the return value of this function is generated depends on whether the animation set is configured to loop or ping pong. For example, imagine that we have called AdvanceTime many times and that the animation controller's track time (the timer of the track the animation set is assigned too) is now set to 155 seconds. Also imagine that the animation set which we are using is configured to loop each time its end is reached. Also, let us assume that this animation has a period of 50 seconds. The animation controller will pass this function the track time (155) and will get returned a value of 5 seconds. This is because at 155 global seconds, the animation set has looped three times and is now on the 5th second of its 4th iteration. Of course, the value returned would be different if the animation set was configured to ping-pong. Ping-Ponging is like looping except, when the animation set reaches the end, the timeline is traversed backwards to the beginning. When the beginning is reached, it starts moving forward again, and so on. Therefore, for every even loop the timeline is moving forward and for every odd loop it is moving backwards (assuming 0 based loop indexing). If we were to call this method on a ping-pong animation, we would get back a periodic position of 45 seconds. This is because it is beginning its fourth cycle and would be moving backwards (5 seconds from its total period of 50 seconds).

The following code might be used by the animation controller when it needed to fetch the SRT data for the first animation in an animation set during a call to AdvanceTime. TrackTime is the current time of the track (accumulated during AdvanceTime calls) which, in our example, is currently at 155 seconds. Do not worry about how the GetSRT function works for the time being; we will cover it in a moment. Just know that it fills the passed scale and translation vectors and the rotation quaternion with the correct interpolated keyframe data given the input local animation set time.

```
D3DXVECTOR3      Scale , Translate; // Used to store animations scale and position
D3DXQUATERNION   Rotate;            // Used to store animations new orientation

AnimationIndex    = 0;   // Get SRT data for first animation in set
TrackTime         = 155; // Accumulated global time of the track.

AnimSetLocalTime  = pAnimSet->GetPeriodicPosition ( TrackTime )
pAnimSet->GetSRT( AnimSetLocalTime, AnimationIndex, &Scale, &Rotate, &Translate );
```

The above code should give you a pretty good idea of the interaction between the animation controller and its animation sets. The important point here is that the animation set's GetSRT function expects to

be given a local time value which is fully contained within the range of time specified by its keyframe data. GetPeriodicPosition performs this mapping from a global time value, giving the animation controller the correct local time value to pass into the GetSRT function.

```
UINT GetNumAnimations(VOID);
```

This function accepts no parameters and returns an unsigned integer containing the number of animations being managed by this animation set. This tells the animation controller (or any caller for that matter) how many frames in the hierarchy are being manipulated by this animation set. Each animation in the animation set contains a frame name and SRT keyframe lists for that frame.

This function will be called by the animation controller during the call to AdvanceTime. The controller can then loop through these animations and call ID3DXAnimationSet::GetSRT for each one. This will generate the new SRT data for each frame in the hierarchy animated by this animation set. This SRT data is then used to build the matrix for each of these frames.

```
HRESULT GetAnimationIndexByName( LPCSTR pName, UINT*pIndex )
```

There may be times when your application wants to fetch the keyframe data for an individual animation within an animation set. You will see in a moment that there are methods of the ID3DXKeyframedAnimationSet interface that allow us to do this, but they must be passed the index of the animation. Often, your application might not know the index but will know the name of the animation/frame. Using this function we can pass in the name of the animation we wish to retrieve an index for and also pass the address of an unsigned integer. On function return, if an animation exists that animates the frame name, the index of this animation will be returned in the pIndex parameter. It is the index of the animation that the animation controller passes to the ID3DXAnimationSet::GetSRT function.

```
HRESULT GetAnimationNameByIndex( UINT Index, LPCSTR *ppName );
```

This function is the converse for the above one. If you know the index of an animation within an animation set, you can use this function to retrieve its name. As the first parameter you pass in the index of the animation whose name you wish to know, and in the second parameter you pass in the address of a string pointer. On function return, the string will contain the name of the animation. As this is also the name of the frame in the hierarchy to which the animation is attached, we can imagine how this could be very useful.

The animation controller itself might well use this function inside its AdvanceTime method. As mentioned previously, the AdvanceTime method will loop through each animation stored in the animation set and call GetSRT for it. Once this SRT data has been combined into a new matrix, that matrix should replace the parent-relative matrix data currently existing in the assigned frame. At this point, the animation controller has the new matrix and the index of the animation it was generated for, but does not yet have its name.

The following code snippet shows how the animation controller might use this function to get the name of the animation which it then uses to find the matching name in the hierarchy. Much of the detail is left out (such as the call to GetSRT for each animation and the construction of the new matrix) but hopefully you will get the overall idea.

```
for ( UINT i = 0 ; i < pAnimSet->GetNumAnimations(); i++ )
{
     …
     …
     // GetSRT would be called here and the new matrix (mtxFrame )
     // for this animation would be  generated
     …
     …
     LPCSTR pName = NULL;
     pAnimSet->GetAnimationNameByIndex( i, &pName );
     pFrame = D3DXFrameFind(m_pFrameRoot, pName );

     // Store the new matrix in the frame hierarchy
     pFrame->TransformationMatrix = mtxFrame;
}
```

The next function, GetSRT, is really the core of the interpolation system.

```
HRESULT GetSRT
(
    DOUBLE            PeriodicPosition,
    UINT               Animation,
    D3DXVECTOR3      *pScale,
    D3DXQUATERNION   *pRotate,
    D3DXVECTOR3      *pTranslate
);
```

This method is usually called by the animation controller to fetch the SRT data for a specified frame in the hierarchy. If no other animation set is being used which has an animation assigned to the same frame in the hierarchy, then this SRT data will be placed directly into the hierarchy frame matrix. Therefore we might say that this method generates the new matrix data for a specified frame in the hierarchy for a specified local time. If multiple animation sets are assigned to the animation mixer which animate the same frame in the hierarchy, then the resulting SRT data for that frame from **each** animation set will be blended together (by the animation controller) based on the current mixer track settings. The resulting blended data will then be placed into the frame matrix.

This function is passed the time (in seconds) in the local timeline of the animation that we would like to have the SRT data returned for. In other words, this function expects to be passed the periodic position in the animation set for which SRT data will be generated for the specified animation. We also pass in an integer as the second parameter describing the animation we would like the SRT data returned for. Remember, an animation is just a named container of keyframe data that is assigned to a frame in the hierarchy. Therefore, if the animation controller (or our application) would like to get the SRT data for a frame in the hierarchy called ‘Wheel_Frame’ for a global time of 406 seconds, we could do something like we see in the following code:

```
UINT              Animation ;
D3DXVECTOR3       Scale , Translate;
D3DXQUATERNION    Rotate;

// Get the index of the frames animation in the animation set
pAnimSet->GetAnimationIndexByName("Wheel_Frame" , &Animation );

// Map ever increasing track time to local time taking into account
// looping or ping-ponging
PeriodicPosition = pAnimSet->GetPeriodicPosition( 405 );

// Get the SRT data for that animation at 405 global seconds
pAnimSet->GetSRT( PeriodicPosition , Animation , &Scale , &Rotate , &Translate);

// Here the animation controller would use Scale, Rotate and Translate to
// build a new frame matrix
```

As the above code demonstrates, if you do not know the index of the animation within the animation set then you can use the GetAnimationIndexByName method to retrieve the index of the animation/frame with the specified name. We can then pass this index into the GetSRT Method. The code above is very similar to the call made by the animation controller to an animation set to fetch the SRT data for each of its animations.

Notice how we also pass in the address of a scale vector, a rotation quaternion and a translation vector. On function return, these three variables will be filled with the SRT data that can then be used to rebuild the matrix.

> **Note** : Our application will not need to call the GetSRT method directly to generate the SRT data for each animation. The animation controller will handle these calls on our behalf when we call ID3DXAnimationController::AdvanceTime. However, you do have access to the interface for each animation set, so your application could use an animation set object as a simple keyframe interpolator even if your application was not using the D3DX animation controller. For example, if you have your own animation system in place, you could manually create a D3DXKeyframedAnimationSet object, fill it with keyframe data and use it to interpolate SRT data for your own proprietary system.

In summary, when we call ID3DXAnimationController::AdvanceTime, the animation controller will loop through each of its active animation sets. For each animation set it will loop through each stored animation. For each animation it will call ID3DXAnimationSet::GetSRT. This function will return the scale, rotation and position that the assigned frame in the hierarchy should have based on the current time. When this process is finished and the SRT data for all animations in all active animations sets is handed back to the controller, it will then check to see if multiple SRT data exists for the same frame in the hierarchy. This can happen when two animation sets have an animation that animate the same frame in the hierarchy. When this is the case, the multiple SRT data sets for that frame are blended together and used to construct the final frame matrix. When only one set of SRT data exists (because a hierarchy frame is being animated by only a single animation set) the SRT data is used to build the frame matrix directly. However, the SRT data will still have the properties of the mixer track to which it is assigned applied to it. Therefore, if the mixer track to which the animation set was assigned was configured to scale the SRT data by 0.5 for example, that scaling process will still occur regardless of whether multiple animations are used or not. We will cover the animation mixer later when we examine the ID3DXAnimationController interface.

So we now know how the ID3DXAnimationSet::GetSRT function is used, but how does it return the SRT data for the requested animation? Although we do not have access to the original DirectX source code, we can, for the purpose of understanding the methodology, write our own version of the function. We will do this at the end of this section. This will demonstrate not only how the GetSRT function works (in case you need to derive your own animation set interfaces for use with the D3DX animation system), but also how it fits into grand scheme of things.

Keyframe Retrieval Functions

The ID3DXKeyframedAnimationSet interface also has several functions that allow an external process to query its underlying keyframe arrays. We can query the number of scale, rotate and translation keys that might exist for a given animation in the set or even retrieve those keyframes for examination. Usually, our application will not need to do this, but these functions can be useful for debugging purposes or for applications that need to manipulate keyframe data directly (like the animation set splitter application we will create in lab project 10.2).

```
UINT GetNumScaleKeys( UINT Animation );
```

This function takes a single parameter, which is the index of the animation we are querying. It returns an unsigned integer describing the number of scale keyframes stored in the animation set for the specified animation. We can use this function to determine how big an array of D3DXKEY_VECTOR3s we need to allocate if we need to retrieve the data (via a call to GetScaleKeys which will be covered in a moment).

```
UINT GetNumRotationKeys( UINT Animation );
```

This function takes a single parameter, which is the index of the animation we are querying. It returns an unsigned integer describing the number of rotation keyframes stored in the animation set for the specified animation. We can use this function to determine how big an array of D3DXKEY_QUATERNIONs to allocate if we need to retrieve the data (via a call to GetRotationKeys which will be covered in a moment).

```
UINT GetNumTranslationKeys( UINT Animation );
```

This function takes a single parameter, which is the index of the animation we are querying. It returns an unsigned integer describing the number of translation keyframes stored in the animation set for the specified animation. We can use this function to determine how big an array of D3DXKEY_VECTOR3s we need to allocate if we need to retrieve the data (via a call to GetTranslationKeys which will be covered in a moment).

HRESULT GetScaleKeys(UINT Animation, LPD3DXKEY_VECTOR3 pKeys);

This function is called to retrieve all scale keyframes stored within the animation set for the specified animation. We must pass the index of the animation we wish to enquire about and pointer to a pre-allocated array of D3DXKEY_VECTOR3 structures large enough to store every scale keyframe. The scale keys are then copied into this array by the function. The following code demonstrates this concept:

```
D3DXKEY_VECTOR3 *pSKeys = NULL;
pSKeys = new D3DXKEY_VECTOR3 [ pAnimSet->GetNumScaleKeys() ];
pAnimSet->GetScaleKeys( pSKeys );

… //Do something with key data here such as read it or copy it….

delete [] pSKeys;
```

Keep in mind that you are getting back a *copy* of the keyframe data, so modifying the data will not affect the behavior of the interpolator.

HRESULT GetScaleKey(UINT Animation, UINT Key,
LPD3DXKEY_VECTOR3 pScaleKeys);

This function is similar to the previous function except it allows us to retrieve only a single specified scale keyframe. We pass in the index of the animation we wish to retrieve the keyframe for, the index of the keyframe in the animation's keyframe array we wish to retrieve, and the address of a D3DXKEY_VECTOR3 structure. If the animation index and the key index parameters are valid, on function return the input structure in the third parameter will contain the scale keyframe at the specified index (UINT Key) in the specified animation (UINT Animation).

HRESULT GetRotationKeys (UINT Animation, LPD3DXKEY_QUATERNION pKeys);

This function is called to retrieve all rotation keyframes stored within the specified animation within the animation set. We must pass in the index of the animation we wish to retrieve the rotation keyframes for, and a pointer to a pre-allocated array of D3DXKEY_ QUATERNION structures large enough to store every rotation keyframe. The rotation keys are then copied into this array by the function.

HRESULT GetRotationKey(UINT Animation, UINT Key,
D3DXKEY_QUATERNION pRotationKeys) ;

This function is the single key version of the previous function. We pass in the animation index, the index of the rotation keyframe we wish to retrieve, and the address of a D3DXKEY_QUATERNION to fill. Remember that this is a copy of the keyframe you are being returned, so modifying it will not alter the behavior of the animation.

```
HRESULT GetTranslationKeys ( UINT Animation, LPD3DXKEY_VECTOR3 pKeys );
```

This function is called to retrieve all translation keyframes stored within the animation set for the specified animation. We pass the index of the animation we wish to retrieve the position keyframes for and a pointer to a pre-allocated array of D3DXKEY_VECTOR3 structures large enough to store every translation keyframe. The translation keys are then copied into this array by the function

```
HRESULT GetTranslationKey( UINT Animation, UINT Key,
                           LPD3DXKEY_VECTOR3 pTranslationKey );
```

This is the single key version of the function previously described. We pass in the index of the animation we wish to retrieve the keyframe for, the index of the translation keyframe we wish to retrieve, and the address of a D3DXKEY_VECTOR3. On successful function return, the vector will contain a copy of the requested key.

Changing Key Properties

The ID3DXKeyframedAnimationSet interface also exposes three methods (one for each keyframe type) that allow the changing of keyframe values on the fly. Each function takes three parameters. The first parameter is the index of the animation you wish to alter the key value for. The second parameter is the index of the key in the animation's S, R, or T lists that you wish to modify. The final parameter is the new key itself. This final parameter should be the address of a D3DXKEY_VECTOR3 structure if you are modifying a scale or translation key, or the address of a D3DXKEY_QUATERNION structure if you are updating a rotation key. These methods are listed next:

```
HRESULT SetRotationKey( UINT Animation, UINT Key,
                        LPD3DXKEY_QUATERNION pRotationKeys );

HRESULT SetScaleKey( UINT Animation, UINT Key,
                     LPD3DXKEY_VECTOR3 pScaleKeys );

HRESULT SetTranslationKey( UINT Animation, UINT Key,
                           LPD3DXKEY_VECTOR3 pTranslationKey );
```

The animation and the key must already exist in the animation set. That is to say, you can not add keys to an animation with these functions; you can only change the values of existing keys.

Removing Keys

Just as we have the ability to change the property of an already defined keyframe, methods are also available that allow for the removal of keyframes from the animation set. Three methods are exposed through which we can unregister scale keys, rotation keys and translation keys respectively. Each of these methods takes two parameters. The first is the index of the animation we wish to unregister a key for, and the second is the index of the key we wish to remove. These methods are shown below.

```
HRESULT UnregisterRotationKey( UINT Animation, UINT Key );

HRESULT UnregisterScaleKey( UINT Animation, UINT Key );

HRESULT UnregisterTranslationKey( UINT Animation, UINT Key );
```

It should be noted that you should not be unregistering keys during the main game loop or in any time critical section of your code. These functions are very slow because they require the animation set to resize the S, R, and T arrays (depending on which key you are unregistering). For example, if you use the UnregisterRotationKey function and specify a key index somewhere in the middle of the array, the array will need to be resized. This involves dynamically allocating a smaller array and copying the data from the old array into the new array (minus the key you just unregistered) before releasing the old key array from memory. Typically you might do something like this just after you have loaded an animated X file and wish to remove certain keyframes that are not needed.

Removing Animations

The ID3DXKeyframedAnimationSet interface also exposes a method that allows for the removal of an entire animation and all of its SRT data from the animation set. This might be useful if you loaded an animated X file that animated a frame in your hierarchy that you did not want animated.

```
HRESULT UnregisterAnimation( UINT Index );
```

This function takes a single parameter; the index of the animation you wish to unregister.

As an example, let us imagine that we loaded an animated X file and found that when playing its animation, one of the frames in the hierarchy that it animated was called 'MyFrame1'. Let us also assume that for some reason we do not wish this frame to be animated by the animation set. After loading the X file, we could use the following code to unregister the animation that animates this frame. The code assumes pAnimSet is a pointer to a valid ID3DXKeyframedAnimationSet interface that has been previously fetched from the animation controller. We will cover how to retrieve interfaces for animation sets when we cover the ID3DXAnimationController method shortly.

```
UINT AnimationIndex;
pAnimSet->GetAnimationIndexByName("MyFrame1", &AnimationIndex );
pAnimSet->UnregisterAnimation ( AnimationIndex );
```

Retrieving the Ticks per Second Ratio

As previously discussed, keyframe timestamps in an animation set are defined internally using arbitrary units known as ticks. How many of these ticks are considered to equal one second in real time is largely dependant on the application which exported the animation data and the resolution the artist used on the timeline. A function is exposed by the animation set which allows the application to query the TicksPerSecond ratio in which its keyframe timestamps are defined.

```
DOUBLE GetSourceTicksPerSecond(VOID);
```

While our application will rarely need to call this method, the animation set object absolutely must know this value. After all, when the animation controller requires the SRT data for a given frame in the hierarchy, it calls the animation set's GetSRT method. This method accepts a time in seconds (not ticks), so the GetSRT function must covert the passed time in seconds into ticks to find the two bounding keyframes to interpolate between.

10.6.2 A Closer Look at GetSRT

In this section we will discuss how we could write our own GetSRT function for an animation set. This will be helpful if you need to derive your own version of an animation set that you will plug into the D3DX animation system. It will also allow us to better understand what the animation controller and its underlying animation sets are doing when we call ID3DXAnimationController::AdvanceTime.

While this source code is not the exact implementation of ID3DXKeyframedAnimationSet::GetSRT, it will return the same data. In fact, we have tested this code as a drop-in replacement and it works very well. This code is for teaching purposes only. Our goal is to look at and understand the interpolation process for SRT keyframes. You will not need to write a function like this in order to use the D3DX Animation System as it is already implemented in ID3DXKeyframedAnimationSet::GetSRT. Note as well that the application will never even have to call this function, because the animation controller will call it for each frame that is animated by each currently active animation set.

As our example function will not be a method of the ID3DXAnimationSet interface, we will add an extra parameter on the end. This will be a pointer to an ID3DXKeyframedAnimationSet interface. We will use this so that our function can fetch the keyframe data from the animation set. In the real method, this pointer is not needed and neither are the methods to fetch the keyframe data from the animation set since the keyframes are owned by the animation set and are directly accessible.

Remember when looking at the following code that this function has one job: to return the SRT data for a single frame/animation for a specified periodic position in the animation set's timeline. We will look at the code a section at a time. As with the function we are emulating, this function should be passed the periodic position and animation index we wish to retrieve the SRT data for, and should also be passed addresses for a scale vector, a rotation quaternion and a translation vector. On function return we will fill these with the interpolated scale, rotation and position information for the specified time. This function will demonstrate what happens behind the scenes when the animation controller calls ID3DXKeyframedAnimationSet::GetSRT for one of its animation sets for a given animation.

```
HRESULT GetSRT( double Time, ULONG Index, D3DXVECTOR3 * pScale,
                D3DXQUATERNION * pRotate, D3DXVECTOR3 * pTranslate,
                ID3DXKeyframedAnimationSet * pSet )
{
    ULONG i;
    double fInterpVal;
    LPD3DXKEY_VECTOR3 pKeyVec1, pKeyVec2;
    LPD3DXKEY_QUATERNION pKeyQuat1, pKeyQuat2;
```

```
    D3DXVECTOR3 v1, v2;
    D3DXQUATERNION q1, q2;

    // Validate parameters
    if ( !pScale || !pRotate || !pTranslate ) return D3DERR_INVALIDCALL;

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

    ULONG ScaleKeyCount       = pSet->GetNumScaleKeys( Index );
    ULONG RotationKeyCount    = pSet->GetNumRotationKeys( Index );
    ULONG TranslationKeyCount = pSet->GetNumTranslationKeys( Index );
```

The first section simply initialises the passed scale and translation vectors to zero and sets the passed quaternion to identity. We then use the passed animation set interface to fetch the total number of scale, rotation, and translation keyframes. As discussed earlier, these three methods take, as their only parameter, the index of the animation we are interested in and return the count values. In the actual ID3DXKeyframedAnimationSet::GetSRT function there would be no need for this call because the keyframe data is stored in the animation set. But our mock function will need to access them in order to simulate the interpolation step.

In the next section we have to convert the passed periodic position into ticks. You will recall from our earlier discussion how each animation set has its keyframe timestamps specified in ticks. As the periodic position passed into the function is in seconds, we must multiply the periodic position by the TicksPerSecond value stored in the X file. You will also recall that there may be an AnimTicksPerSecond data object in the X file describing the timing granularity for the timestamps exported by the modelling application. If none is specified, then a default value of 4800 ticks per second is used. The animation set internally stores this ticks per second value. We can use another ID3DXKeyFramesAnimationSet interface method called ‘GetSourceTicksPerSecond’ to retrieve the number of ticks per second. By multiplying the periodic position (in seconds) by the ticks per second, we get the periodic position in ticks. This is the timing value used to define the keyframe timestamps.

```
    double TicksPerSecond     = pSet->GetSourceTicksPerSecond( );
    double CurrentTick        = Time * TicksPerSecond
```

Notice how in the above code we store the period position converted into ticks in the CurrentTick local variable.

As our function is not a method of ID3DXKeyframedAnimationSet and therefore does not have direct access to its keyframe arrays, the following code will need to fetch the keyframes from the animation set. We therefore allocate the (possible) three keyframe arrays (scale, rotation and translation) using the count values previously fetched, and then pass these arrays into the GetScaleKeys, GetTranslationKeys and GetRotationKeys to fetch copies of the keyframe arrays for the specified animation. Remember, the index of the animation we wish to fetch was passed in as the ‘index’ parameter.

```
    D3DXKEY_VECTOR3 * ScaleKeys = NULL;
    D3DXKEY_VECTOR3 * TranslateKeys = NULL;
    D3DXKEY_QUATERNION * RotateKeys = NULL;
```

```
    if(ScaleKeyCount )
       ScaleKeys = new D3DXKEY_VECTOR3[ ScaleKeyCount ];

    if(TranslationKeyCount)
      TranslateKeys= new D3DXKEY_VECTOR3[ TranslationKeyCount ];

    if( RotationKeyCount )
      RotateKeys = new D3DXKEY_QUATERNION[ RotationKeyCount ];

    if ( ScaleKeyCount )      pSet->GetScaleKeys( Index, ScaleKeys );
    if ( TranslationKeyCount )pSet->GetTranslationKeys( Index, TranslateKeys );
    if ( RotationKeyCount )   pSet->GetRotationKeys( Index, RotateKeys );
```

At this point, we have the keyframe data for the animation we wish to generate interpolated SRT data for. Now it is time to examine each array in turn and perform the interpolation. We will start by looking at the generation of the interpolated scale data.

In order to perform keyframe interpolation, we must find the two keyframes that bound the current tick time. Therefore, we loop through each scale key searching for the correct pair. If we find two consecutive keyframes where the current tick is larger or equal to the first and smaller or equal to the second, then we have found our bounding keyframes. When this is the case, we store these two keyframes in pKeyVec1 and pKeyVec2 and break from the loop.

```
    // Calculate an interpolated scale by finding the
    // two key-frames which bound this timestamp value.
    pKeyVec1 = pKeyVec2 = NULL;
    for ( i = 0; i < ScaleKeyCount - 1; ++i )
    {
        LPD3DXKEY_VECTOR3 pKey     = &ScaleKeys[i];
        LPD3DXKEY_VECTOR3 pNextKey = &ScaleKeys[i + 1];

        // Do these keys  bound the requested time ?
        if ( CurrentTicks >= pKey->Time && CurrentTick <= pNextKey->Time )
        {
            // Store the two bounding keys
            pKeyVec1 = pKey;
            pKeyVec2 = pNextKey;
            break;

        } // End if found keys
    } // Next Scale Key
```

At this point, we have the two scale keyframes which must be interpolated. Next we copy the 3D scale vectors from each keyframe (stored in D3DXKEY_VECTOR3::Value) into 3D vectors v1 and v2. We have discussed interpolation several times in prior lessons and this interpolation is no different. We calulate the time delta between the current tick and the first keyframe's timestamp, and then we divide that by the time delta between both timestamps. This returns a value in the range of 0.0 to 1.0 which can be used to linearly interpolate between the two 3D vectors. The result of the linear interpolation is stored in 'pScale' so that it is accessible to the caller on function return.

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

```
    {
        // Interpolate the values
        v1 = pKeyVec1->Value;
        v2 = pKeyVec2->Value;
        fInterpVal  = CurrentTick - pKeyVec1->Time;
        fInterpVal /= (pKeyVec2->Time - pKeyVec1->Time);
        D3DXVec3Lerp( pScale, &v1, &v2, (float)fInterpVal );
    } // End if keys were found
```

Notice in the above code that we use the 3D vector interpolation function provided by D3DX. D3DXVec3Lerp performs the interpolation between 3D vectors v1 and v2 using the passed interpolation percentage value. This function simply does the following:

V1 + fInterpVal * (V2 – V1)

As fInterpVal is a value between 0.0 and 1.0, we can see how the resulting vector will be a vector positioned somewhere between these two keyframes. We have now successfully generated the S component of the SRT data that the function needs to return.

Our next task is to interpolate the rotation component of the SRT data. We do this is very much the same way. We first loop through every keyframe in the rotation keyframe array (remember that these are stored as quaternions) and search for the two keyframes the bound the current tick.

```
    // calculate an interpolated rotation by finding the
    // two key-frames which bound this timestamp value.
    pKeyQuat1 = pKeyQuat2 = NULL;
    for ( i = 0; i < RotationKeyCount - 1; ++i )
    {
        LPD3DXKEY_QUATERNION pKey     = &RotateKeys[i];
        LPD3DXKEY_QUATERNION pNextKey = &RotateKeys[i + 1];

        // Do these keys bound the requested time ?
        if ( CurrentTick >= pKey->Time && CurrentTick <= pNextKey->Time )
        {
            // Store the two bounding keys
            pKeyQuat1 = pKey;
            pKeyQuat2 = pNextKey;
            break;
        } // End if found keys
    } // Next Rotation Key
```

Now that we have found the two quaternions that describe the rotation to interpolate, we will interpolate between the two quaternions using the D3DXQuaternionSlerp method. Performing a spherical linear interpolation (SLERP) is one of the main reasons quaternions are used to represent rotations. Rather than linearly interpolate between the rotation angles which would cut a straight line from one to the other, a spherical linear interpolation allows us to interpolate along an arc between the rotations. This provides much smoother and more consistent movement from one rotation to the next. If we imagine that the two rotation vectors describe two points on a sphere, slerping between them will follow the path from one point to the next over the surface of the sphere. A linear interpolation would cut a straight line through the sphere from one point to the other.

```
    // Make sure we found keys
    if ( pKeyQuat1 && pKeyQuat2 )
    {
        // Interpolate the values
        D3DXQuaternionConjugate( &q1, &pKeyQuat1->Value );
        D3DXQuaternionConjugate( &q2, &pKeyQuat2->Value );
        fInterpVal  = CurrentTick - pKeyVec1->Time;
        fInterpVal /= (pKeyVec2->Time - pKeyVec1->Time);
        D3DXQuaternionSlerp( pRotate, &q1, &q2, (float)fInterpVal );
    } // End if keys were found
```

Notice how the above code converts the quaternions for compliance with a left-handed coordinate system by using the D3DXQuaternionConjugate function. This function negates the sign of the *x,y,z* components of the quaternion (the rotation axis), essentially flipping it to point in the opposite direction. The reason we do not need to negate the *w* component (the rotation angle) also is that the D3DX functions expect the *w* component of the quaternion to represent a counter-clockwise rotation about the axis. This is the opposite direction that rotations are interpreted in a matrix. Therefore, by flipping the axis and not changing the sign of *w*, we essentially end up with the rotation axis flipped so that it now represents the same axis in a left-handed coordinate system with a *w* component giving us a clockwise rotation about this axis. Note as well how the result of the slerp is stored in the quaternion structure passed into the function so that it is accessible the caller on function return.

Finally, we interpolate the translation vector using the two bounding keyframes. This code is almost identical to the code that interpolates the scale vectors, so it is shown in its entirety below.

```
    // Calculate an interpolated translation by finding the
    // two key-frames which bound this timestamp value.
    pKeyVec1 = pKeyVec2 = NULL;
    for ( i = 0; i < TranslationKeyCount - 1; ++i )
    {
        LPD3DXKEY_VECTOR3 pKey     = &TranslateKeys[i];
        LPD3DXKEY_VECTOR3 pNextKey = &TranslateKeys[i + 1];

        // Do these keys bound the requested time ?
        if ( CurrentTick >= pKey->Time && CurrentTick <= pNextKey->Time )
        {
            // 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 )
    {
        // Interpolate the values
        v1 = pKeyVec1->Value;
        v2 = pKeyVec2->Value;
        fInterpVal  = CurrentTick - pKeyVec1->Time;
        fInterpVal /= (pKeyVec2->Time - pKeyVec1->Time);
        D3DXVec3Lerp( pTranslate, &v1, &v2, (float)fInterpVal );
    } // End if keys were found
```

Finally, with the SRT data correctly interpolated and stored correctly, this version of the function must delete the key arrays that were temporarily allocated to retrieve the keyframe data from the animation set. Obviously, the ID3DXKeyframedAnimationSet::GetSRT method would not need to allocate or deallocate these arrays as it would have direct access to the keys.

```
    if ( ScaleKeys ) delete []ScaleKeys;
    if ( RotateKeys ) delete []RotateKeys;
    if ( TranslateKeys ) delete []TranslateKeys;

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

There are still a few methods of the ID3DXKeyframedAnimationSet that we have not yet covered. Primarily, the ones we are most interested in for this chapter are those we will use to register callback functions that are triggered by keyframes. We will look at these methods later when we cover the D3DX animation callback system.

10.6.3 D3DXCreateKeyFramedAnimationSet

There will be times when you will be unable to use D3DXLoadMeshHierarchyFromX to load animation data. For example, you may be creating runtime animation data for the animation controller or perhaps loading a custom animation file. In these cases you will need to know how to manually build D3DXAnimationSet objects so that you can register them with the animation controller yourself. D3DX exposes a global function for this purpose called D3DXCreateKeyFramedAnimationSet. The function is shown below with its parameter list followed by a description of each parameter.

```
HRESULT WINAPI D3DXCreateKeyframedAnimationSet
(
    LPCSTR pName, DOUBLE TicksPerSecond,
    D3DXPLAYBACK_TYPE Playback,
    UINT NumAnimations, UINT NumCallbackKeys,
    CONST LPD3DXKEY_CALLBACK *pCallKeys,
    LPD3DXKEYFRAMEDANIMATIONSET *ppAnimationSet
);
```

Let us discuss the parameters to this function one at a time:

LPCSTR *pName*

This function allows you to assign the animation set a name. While not required it is often more intuitive to work with named animation sets. You will see when we cover the ID3DXAnimationController interface shortly, that one of the things we will often want to do is have it return us a pointer to one of its animation sets. While we can fetch animation sets by index, it is often preferable to use ID3DXAnimationController::GetAnimationSetByName function (covered shortly). It is certainly more intuitive when looking at animation code to see references to animation sets such as ‘Walk’ or ‘Run’ instead of indices like 2 or 4. This parameter is analogous to the name that may be assigned to an AnimationSet data object in an X file.

DOUBLE *TicksPerSecond*
When creating an animation set manually, we will be responsible for filling its SRT keyframe arrays. Therefore, the units of measurement used for keyframe timestamps (ticks) are completely up to us. We must inform the animation set, at creation time, the number of ticks per second that we will be using in our keyframe timestamps. This is needed because the GetSRT method (called by the animation controller) is passed a periodic position in seconds which must be converted to ticks. This is the means by which correct bounding keyframes are found.

D3DXPLAYBACK_TYPE *Playback*
This parameter is used to inform D3DX how we would like the mapping between global time (the time of the track the animation set is assigned to) and the animation set's local time (the periodic position) to be performed. You will recall that before the animation controller calls ID3DXAnimatonSet::GetSRT, it must first map the global track time the animation set is assigned to into the animation set's local time using the ID3DXAnimationSet::GetPeriodicPosition function. It is this function that returns the local animation time that is passed into GetSRT. With this parameter we can choose how this function calculates the periodic position of the animation set and thus how the animation set behaves when the track time exceeds the animation sets total period.

For example, we can configure the GetPeriodicPosition method such that if the track time passed in exceeds the period, the periodic position is wrapped around, causing the animation set to continuously loop as the global time climbs ever higher. Therefore our application can repeatedly call ID3DXAnimationController::AdvanceTime without worrying about whether the accumulated global time of the controller (and each of its tracks) is inside or outside the period of the animation set.

Alternatively, we can choose for the mapping to occur such that if the global time exceeds the period of the animation, the animation set essentially stops playing (remains frozen at its last keyframe). Using this mapping, as soon as the track time exceeds the period of the animation set assigned to that track, the animation set stops playing.

Finally, we can make the GetPeriodicPosition method map the track time to a periodic position using ping-ponging. As discussed previously, when this mapping is used, when the animation set period is exceeded, its periodic position starts to backtrack towards the beginning of the local timeline. When the beginning is reached, the timeline direction is flipped again so that the periodic position starts moving forwards towards the end of the timeline, and so on.

We can specify the three possible behaviors of the animation set we are creating using a member of the D3DXPLAYBACK_TYPE enumeration:

```
typedef enum _D3DXPLAYBACK_TYPE {
    D3DXPLAY_LOOP = 0,
    D3DXPLAY_ONCE = 1,
    D3DXPLAY_PINGPONG = 2,
    D3DXEDT_FORCE_DWORD = 0x7fffffff
} D3DXPLAYBACK_TYPE;
```

To better understand how the three modes cause the GetPeriodicPosition function to map global track time to periodic positions differently, let us examine the case of an animation

set that has a period of 25 seconds. Let us also assume that the track time we wish to map to a periodic position (to feed into the GetSRT function) is currently at 30 seconds.

D3DXPLAY_LOOP
If the animation set is created with this option, then when a global track time of 30 seconds is reached by the animation controller, the GetPeriodicPosition method of the animation set will return a periodic position of 5 seconds. This is because the animation set has reach the end and is now five seconds (30 – 25) into its second loop.

D3DXPLAY_ONCE
If the animation set is created with this option, then after 25 seconds the animation would stop playing. The frame matrix would be essentially frozen in the position specified by its final SRT keyframes. At 30 seconds of global track time, the animation would have stopped playing 5 seconds earlier.

D3DXPLAY_PINGPONG
If the animation set is created with this option, then at 30 seconds global track time, the GetPeriodicPosition method would return a periodic position of 20 seconds. This is because the period of the animation set was reached 5 seconds earlier (at 25 seconds) and has backtracked 5 seconds from the end towards the beginning (25 – [30-25]).

UINT *NumAnimations*
When we create an animation set we must specify the maximum number of frames in the hierarchy it should be capable of containing animations for. As previously discussed, each animation contains a frame name and SRT keyframe arrays for that frame. Although we are allocating room for the maximum number of animations at animation set creation time, the keyframe data will be added to each animation as a separate step once it has been created. We shall see this in a moment.

UINT *NumCallbackKeys*
CONST LPD3DXKEY_CALLBACK **pCallKeys*
These parameters will be covered later in the chapter when we discuss the D3DX animation system's callback mechanism. Just know for now that you have the ability to pass in a series of callback keys. Each key contains a timestamp and a collection of data. When one of these callback keys is encountered in the timeline during an animation update, this data can be passed to an application defined callback function that performs some arbitrary task (e.g. playing a sound effect or disabling the animation track). Callback keys are defined per animation set instead of per animation. While they too are defined using ticks, they are handled quite differently by the controller and animation set. For this reason, we will forget about them for now and revisit them later. For now, let is just assume that we will pass in NULL as these two parameters. This informs D3DX that this animation set will not have any callback events registered with it.

LPD3DXKEYFRAMEDANIMATIONSET **ppAnimationSet*
As the final parameter to this function we must pass in the address of an ID3DXKeyframedAnimationSet interface pointer. On successful function return, it will be a pointer to the interface for the newly created animation set object.

An important thing to note is that in order for animation sets to be used, they must be *registered* with an animation controller. This is because the ID3DXAnimationController::AdvanceTime function ultimately controls all animation processing. If the animation set is not registered with the animation controller, the animation controller will have no knowledge of the animation set and will not use it. Therefore, we can think of this function as returning an animation set that is currently a detached object (i.e., not yet plugged into the animation subsystem). We will shortly discuss how to manually create an ID3DXAnimationController and how to register animation sets with it, but for now you can rest assured that when we use D3DXLoadMeshHierarchyFromX and specify an X file that contains animation data, all of this will be handled for us automatically.

Note as well that when D3DX loads an X file that contains multiple animation sets, although all animation sets will be registered with the returned animation controller, only the last animation set defined in the X file will be assigned to an active track on the animation mixer (track 0). This is why we do not have to configure the animation controller passed back to our application in Lab Project 10.1. The X file in that project contains only a single animation set and it is automatically assigned to track 0 on the animation mixer by the D3DX loading function. Thus it is ready for immediate playback. If an X file contains multiple animation sets, and the last animation set is not the first one you would like to playback, you can use the ID3DXAnimationController interface to assign a different animation set to that track (or to another enabled track if you intend to do animation blending).

Manually Populating the Animation Set

After a keyframed animation set has been manually created, it is initially empty and contains no SRT data. The next task of the application will usually be to add the SRT data for each frame in the hierarchy that this animation set intends to animate. We call the ID3DXKeyframedAnimationSet::RegisterAnimationSRTKeys method to place the SRT data for a single frame in the hierarchy into the animation set. In other words, if you wanted your animation set to animate five frames in your hierarchy (five animations), then you would call this method five times. Each call would specify the SRT data for a single frame. The method is shown below with a description of each of its parameters.

```
HRESULT RegisterAnimationSRTKeys
(
    LPCSTR pName,
    UINT NumScaleKeys,
    UINT NumRotationKeys,
    UINT NumTranslationKeys,
    CONST LPD3DXKEY_VECTOR3 *pScaleKeys,
    CONST LPD3DXKEY_QUATERNION *pRotationKeys,
    CONST LPD3DXKEY_VECTOR3 *pTranslationKeys,
    DWORD *pAnimationIndex
);
```

LPCSTR pName
This parameter is how we inform the animation set which frame in the hierarchy the keyframe data we are passing in is intended for. This string should contain the name of a frame in the hierarchy and thus

this also becomes the name of the animation itself. This is the animation name you would use in calls to methods like ID3DXAnimationSet::GetAnimationIndexByName if you wished to retrieve the index of the animation within the animation set's internal animation array.

UINT NumScaleKeys
UINT NumRotationKeys
UINT NumTranslationKeys
These three parameters (any of which may be zero) inform the animation set how many separate scale, rotation and translation keyframes we are going to register with the animation set for this animation. This allows the function to allocate the internal SRT arrays of the animation to the requested size. It also informs the function how many keys to expect in the arrays we pass in using the next three parameters.

CONST LPD3DXKEY_VECTOR3 ***pScaleKeys***
CONST LPD3DXKEY_QUATERNION ***pRotationKeys***
CONST LPD3DXKEY_VECTOR3 ***pTranslationKeys***
These three parameters are used to pass in the already allocated and populated arrays of the scale, rotation and translation keyframes. The application should have allocated these arrays and populated them with keyframe data prior to registering them with the animation set. The animation set will copy these keyframes into its internal SRT arrays and thus, these arrays may be released from memory after the function has returned. Any of these pointers can be set to NULL, indicating that there is no keyframe data of that type. For example, if the pScaleKeys parameter is set to NULL then this animation does not contain any scale animation data. You should also make sure that if this is the case, the NumScaleKeys parameter is also set to zero.

DWORD ***pAnimationIndex***
For the final parameter we can pass in the address of a DWORD which on function return will contain the index of the animation we have just created. The application may wish to store this away and use it later to access the animation keys without having to first call GetAnimationIndexByName to retrieve the index first. You may pass NULL as this parameter if you do not wish to store the index of the animation about to be created.

To better understand the creation process, the following example code will create an ID3DXKeyframedAnimationSet and will add animation data to it for a frame in the hierarchy called 'MyFrame'. In this simple example, the animation data will contain just two scale keys at timestamps of 0 and 2000 ticks. This animation set will be created with a ticks per second ratio of 100. This means the first keyframe will be executed at 0 seconds and the second keyframe will be executed at 20 seconds (2000 / 100).

```
// Create new keyframed anim set called "Anim1" with a 100m ticks per second
// It is looping, will contain 1 animation and will have 0 callback keys (0,NULL)
ID3DXKeyframedAnimationSet * pAnimSet = NULL;
D3DXCreateKeyFramedAnimationSet("Anim1",100,D3DXPLAY_LOOP, 1, 0, NULL, &pAnimSet);

// Let us now allocate an array of two scale keys. At time zero the frame matrix
// is scaled by 5 and at time 2000 ( 20 seconds ) the matrix is scaled 10x
D3DXKEY_VECTOR3 ScaleKeys[2];
ScaleKeys[0].Time = 0;
ScaleKeys[0].Value= D3DXVECTOR3( 5 , 5 , 5);
ScaleKeys[1].Time = 2000;
```

```
ScaleKeys[1].Value= D3DXVECTOR3( 10 , 10 , 10);

// Now add a new animation to the animation set for 'MyFrame'. It has:
// 2 scale key, 0 rotation keys, 0 translation keys. We pass in the ScaleKeys
// array and pass NULL for the other two key types. We also pass in NULL as the
// animation index which we do not require in this example.
pAnimSet->RegisterAnimationSRTKeys("MyFrame",2,0,0,ScaleKeys,NULL,NULL,NULL);
```

At this point we have an animation set with keyframe data for a single frame in the hierarchy but it is still not plugged into the D3DX animation system. For that to happen we must register the animation set with the animation controller (covered shortly). Once that is done, we can assign it to tracks on the animation mixer and play it back in response to calls to ID3DXAnimatonController::AdvanceTime by the application.

In the next section we will finally discuss the ID3DXAnimationController interface. While it exposes a great many methods (thus making it very flexible), do not feel intimidated. We have already learned that using the animation controller can be very straightforward for basic animation tasks (see Lab Project 10.1). At its simplest, we can call a single method (AdvanceTime) to play our animations. However, we can do many more exciting things with this powerful interface, so let us begin to explore how it works.

10.7 The ID3DXAnimationController Interface

As discussed in the last section, there may be times when D3DXLoadMeshHierarchyFromX is not an option available to us for loading animation data. In those cases, we will have to tackle things manually. Just as there are D3DX functions to manually create keyframed animation sets, there is also a global D3DX function that allows us to create an animation controller to manage them. This function is called D3DXCreateAnimationController. Examining the parameter list to this function will help us to better understand the various internal components of animation controller objects and how they work together. So we will break with tradition and cover the manual creation of the animation controller first. But before we cover this function and its parameters, let us first briefly review, at a high level, what the D3DXAnimationController is and what its responsibilities are in the animation system. This is the interface we are returned from D3DXLoadMeshHierarchyFromX and often is the only interface in the D3DX animation system that our application will work with directly. You are reminded that you do not need to create an animation controller manually if you are loading the data from an X file (D3DX creates the animation controller for you and returns it from the loading function). However there may be times when the animation controller created by D3DXLoadMeshHierarchyFromX is not suitable for our application's needs. In such cases, we will need to know how to create or clone a new controller.

Animation Set Manager

Just as the ID3DXKeyframedAnimationSet can be thought of as an animation manager, the ID3DXAnimationController can be thought of as an ID3DXAnimationSet manager. The

ID3DXAnimationController interface can be used to assign any of its currently managed animation sets to tracks on its internal animation mixer. These tracks can be played back individually or blended together to create complex animation sequences. Each track on the animation mixer has several properties which can be set (speed, position, weight, etc.) to influence the way a given animation set assigned to the track plays back at runtime.

Manually creating an animation controller starts us off with an empty container (much like manually creating an animation set). On its own it is not much use because it does not currently have any animation data assigned to it for management purposes. So the first thing we will usually do after creating an animation controller is call its RegisterAnimationSet method to register various (already created) animation sets. Once an animation set has been registered with the animation controller, it will need to be assigned to a mixer track if we intend to use it during playback to influence the matrices in our frame hierarchy. From there, any calls to AdvanceTime will cause the animation set to update the appropriate nodes in the hierarchy as defined by the artist/animator.

Animation Set Mixer

Animation set registration does not activate the animation -- the animation set must be assigned to an active mixer track for the pre-recorded sequence to be played. When an application calls ID3DXAnimationController::AdvanceTime, only currently active tracks on the animation mixer will have their periodic positions updated. Actually, it is the track the animation set is assigned to which has its own local time which is updated and not the animation set itself. It is the track's time that is used to generate the periodic position of the animation set which in turn is used to generate the SRT data. Therefore, when we call the AdvanceTime method of the animation controller, this elapsed time is also added to the local clocks of each track. By using the track time to generate the periodic position of its assigned animation set, we have the ability to forward or rewind the timeline of individual animation sets on their tracks.

While an animation controller might have many animation sets registered with it, it might currently have only one set assigned to an active track. This would be the only set used to update the hierarchy during the AdvanceTime call. This approach allows the application to play different animation sets at different times simply by assigning animation sets to active tracks when they need to be played. If we have multiple animation sets assigned to active tracks on the mixer, all of those animation sets will affect their respective frames in the hierarchy simultaneously. The final transformation for a given frame in the hierarchy will be the blended result from each animation in each active animation set that is linked to that specific frame.

The ID3DXAnimationController interface has many functions that govern how animation sets are assigned to the various tracks on the mixer and how much weight those sets will contribute to the final blended result. Animation set weighting allows us to play, for example, two animations that might both manipulate some of the same frames in the hierarchy, but allows one set to have more influence over the final result than the other. For example, the first animation set might influence a given frame at 75% of its full weight while the second set only influences the same frame at 25% of its full weight. So while both animation sets have influenced the hierarchy, there was a 3:1 influence ratio in favor of the first set (thus the animations stored in set one would be more visually dominant than those in set two).

Frame Matrix Updates (Animation Outputs)

We have said on a number of occasions that the animation controller uses its animations to manipulate frames in the hierarchy. In truth, the animation controller is really only concerned with the frame matrices themselves for a given named animation. In truth, we could connect them to any arbitrary matrices we choose -- not necessarily only matrices in a frame hierarchy. We will see momentarily that the animation controller maintains a list of pointers to all the frame matrices in the hierarchy that it needs to animate. Internally these matrix pointers are stored in an array so that the animation controller can quickly access them when it needs to rebuild them (when AdvanceTime is called). These matrices that will receive the final SRT output from the mixer are referred to as "Animation Outputs" by D3DX. Thus, as we will see in the next section when we build an animation controller manually, in addition to registering all of the animation sets with the controller, we must also register all of the output matrices in the hierarchy that the animation controller will need to update, so that it has access to those matrix pointers. If a frame's matrix is not registered with the animation controller, then it cannot be manipulated by any of its registered animations (even though those animations are linked to frames in the hierarchy by name). In fact, when we register a matrix with the animation controller, we must also give it a name. This matrix name must match the name of the animation that will be used to animate that frame.

Of course, all of this is handled on our behalf if we use D3DXLoadMeshHierarchyFromX. D3DX will assign each animation within an animation set the name of the frame that it animates (if one exists) and the matrix of that frame will also be registered with the animation controller using the same name as the animation. If we have a frame in our hierarchy called 'RotorBlades', then when D3DX creates the animations(s) that animate this frame, they too will be assigned the name 'RotorBlades'. Furthermore, the transformation matrix of that frame will be registered as an animation output in the animation controller's matrix table and assigned the name 'RotorBlades' as well. This clearly establishes the relationship between the matrix and the animation which alters its value with animation results.

When we manually create our own animation controllers, we will register frame matrices using the ID3DXAnimationController::RegisterAnimationOutput function. It is important to note however that because of the nature of hierarchical data structures (any changes to a matrix will affect all child frame matrices) we must register *all* matrices that the animation controller needs to rebuild. This means that in our example, we must not only register the transformation matrix of our RotorBlade frame, but all child frame matrices of the RotorBlade frame as well. This allows the animation controller to correctly update all affected matrices. While this sounds rather tedious, fortunately there is a global D3DX function to handle this matrix registration with a single function call -- it is called D3DXFrameRegisterNamedMatrices and we will look at it a little later. Basically, we just pass this function a pointer to the root frame of our hierarchy and a pointer to our animation controller interface, and it will step through the hierarchy and register all of the matrices that belong to 'named' frames in the hierarchy. Thus we can manually register all of our matrices with the animation controller with a single function call.

> **Note:** D3DXLoadMeshHierarchyFromX will automatically create the animation controller and register all animation sets and matrices. This means that the returned animation controller interface can be used immediately for hierarchy animation as seen in Lab Project 10.1.

You might have a concern about the dependency of the system on names, given our earlier discussion about the optional nature of frame names in X files. Not to worry however. While it is true that an X file frame data object may not be assigned a name, this will not be the case if that frame is being animated. Remember that inside the X file the Animation data object (i.e. keyframe data) is linked to a frame by embedding the frame's name inside the Animation object as a data reference object. If the frame did not have a name, it could not be bound to any animation inside the X file either. So we can be confident that animated frames within the X file will always have assigned names. D3DX will thus have what it needs to register both the animation assigned to a frame and its matrix with the animation controller, to sustain the link between them.

Memory Management

Animation controllers have a number of restrictions that can be imposed on them by the application or by D3DX at creation time. These limitations allow us to create more memory-friendly animation controllers that conserve space. For example, if we only plan to use one animation set at a time, then we know that we only need to use one track on the animation mixer. It would be wasteful if the mixer was always created with 128 tracks for example. Likewise, it would be wasteful for the animation controller to be created with a matrix pointer array large enough to animate a maximum of 1000 matrices if we only wish to animate a single matrix in our hierarchy. The same is true with regards to how many animation sets must be set up for management. If we know that we will only register four animation sets in total with the controller, it would be wasteful to create an animation controller that was capable (at the cost of some internal memory overhead) of managing 20 animation sets.

While these figures are arbitrary, the important point is that when an animation controller is created manually (using the D3DXCreateAnimationController function), such size restrictions are set at this point and remain fixed for the lifetime of the controller. In the case of the D3DXLoadMeshHierarchyFromX function, D3DX will create the animation controller for us, so setting these limitations is out of our hands. We will see in a moment exactly what the limitations of the animation controller created by D3DX are when we cover manually creating one.

While these restrictions are fixed for a given controller after creation, the ID3DXAnimationController does include a Clone function which allows us to create a new animation controller from an existing one (such as the one D3DX created on our behalf). Much like we saw in Chapter Eight when discussing the cloning of meshes, when we clone a new controller we will be able to extend the capabilities of the original controller. Therefore, if the capabilities of the animation controller that D3DXLoadMeshHierarchyFromX created for us did not meet our requirements, we can clone the animation controller into a version that does. The new controller would retain all of the registered animation sets and keyframe data, so we do not have to rebuild or re-register them. This allows us to make a copy of the animation controller but extend its features (ex. increase the maximum number of tracks or maximum number of matrices, etc.) much as we did with meshes in Chapter Eight.

Sequencing

The animation controller also implements an event scheduler referred to as the sequencer. Using methods exposed by the ID3DXAnimationController interface, we can set events on the controller's *global* timeline which modify the properties of a specific mixer track (or its clock) when that event is triggered. When we register events with the sequencer, we must make sure that we understand which timeline those events are being scheduled for. For example, we know that every time we call AdvanceTime, the elapsed time passed into the function is added to the global time of the animation controller. Not an awful lot happens with the global time of the animation controller if sequencing is not being used. Essentially it is just a running total of elapsed time. To be clear, this global time is *not* the time used to map to a periodic position in an animation set. That is the track time. We discussed earlier in this chapter that when we call AdvanceTime and the global time of the controller is updated, the same elapsed time is also added to the timers of each active track on the mixer. It is the timer/clock of a track that is used to map to a periodic position in its animation set prior to any GetSRT calls.

It may seem strange at first for the controller to maintain a global timer and a timer for each track when essentially the same elapsed time passed into the AdvanceTime function is added to all of the timers simultaneously. If this is the case, then it would seem that the global time of the controller and the timers of each track would always have the same time as one another. After all, they all start off at zero and have the same elapsed time added to them every time AdvanceTime is called by the application. While this is certainly true if we are performing nothing more complex than simple AdvanceTime calls, we will see that we can decouple the global timer from the track timers by adjusting the properties of each track. The sequencer allows us to schedule events in global time that will alter the properties of given track or even its timer. For example, we could set an event that is triggered at a global time of 10 seconds which disables a specific track and its assigned animation set. We could also set another event at 20 seconds (global time) that would re-enable that same track. At this point, the timer of the track would be different from the global timer because it was not being updated along with the global timer after it was disabled. Thus, in this scenario, when the global time has accumulated to 25 seconds, the track (which was disabled for 10 seconds) would have a timer that contains 15 seconds of accumulated time. Again, it is the 15 second track time that would be mapped into a periodic position and used to fetch the SRT data of animations in its assigned animation set.

Another example demonstrating that sequencer events are added to the global timeline is one in which we intend to alter the timer of a track at a specific global time. For example, imagine we have an animation set with a period of 20 seconds. Let us also imagine that we set an event on the sequencer at 20 seconds that will set the timer of the track back to zero. Let us also assume that this animation set is not configured for looping, that is to say, it is designed to play through only once.

In this scenario, the application would call AdvanceTime repeatedly, thus updating both the global time of the controller and the track time of the animation set by the same amount. At 20 seconds global time, an event is found on the global timeline that alters the track timer of our animation set back to zero. At this point, the track time would be zero and the global time would be 20. At 30 seconds global time, the animation set would be 10 seconds into its second iteration even though the animation set was not configured to loop. This is because we manually set the track timer back to zero at a global time of 20 seconds, which starts that track playing from the beginning again. As far at that animation set is

concerned, it is like it is playing for the first time. What would happen at a global time of 40 seconds? Would the track, having reached the end of its period for a second time, loop back to the beginning again? No, it would not. We have already said that the animation is not set to loop and as we have no event set on the sequencer for 40 seconds, no action is taken. At 41 seconds, the track time will be set at 21 seconds. When the track time of 21 is mapped to the periodic position of the animation set, it is found to have exceeded its period. Since the animation set is not configured to loop, at 21 seconds (track time) the animation has finished.

Finally, imagine that we had also scheduled an event that reset the track time back to zero again at a global time of 100 seconds. When a global time of 100 seconds was reached, the animation would start to run from the beginning again (once) for the next twenty seconds, until the track timer once again exceeded 20 seconds (the period of the non-looping animation set). Therefore, for the global time period 100-120 seconds, the track timer was once again in the range 0-20 seconds.

Other possible sequencing events include the ability to alter the playback speed of a given animation set at a particular moment, and the ability to dynamically change a mixer track blending weight at a certain global time so that the animation set assigned to that track can contribute more or less to the final result at the given timestamp.

> **Note**: Sequencer events are registered with the controller's global timeline. Even if an animation set is set to loop or ping-pong, the event will only ever be triggered once, when the global time passes the event on the timeline. This is true even if we reset the track timer. If we had an event scheduled at 20 seconds on the global timeline which reset the timer of a track to zero, the event would not be triggered again when the same track time reaches 20 seconds for a second time. This is because we are only resetting the track time and not the global time. The global time continues to increase with each call to AdvanceTime and would at this point be at 40 seconds. We will see in a moment how we can also reset the global time.

As with all other capabilities, when we create an animation controller, it will have a maximum number defining how many sequencing events can be simultaneously set. When we create the animation controller manually, we can set this value according to the needs of our application. But when using D3DXLoadMeshHierarchyFromX (where the animation controller is created on our behalf), the maximum will be 30 key events scheduled at any one time. If we need to schedule more key events than this, then we will have to clone the animation controller into one with greater event management capabilities. Do not worry; using the sequencer will be explained in detail momentarily. Just know for now that it exists to provide us the ability to script changes for mixer tracks at specific global times.

10.7.1 D3DXCreateAnimationController

Now that we have a basic idea of what the animation controller has to offer, we can begin to examine the specifics. While our application will not normally need to create an animation controller from scratch (because D3DXLoadMeshHierarchyFromX will do it for us), it helps to examine the process since it provides good insight into the internal data layout and inner workings of the animation controller. Remember that if you are loading your animation data from a custom format and cannot use the D3DXLoadMeshHierarchyFromX function, you will also need to manually build your keyframe animations and animation sets in addition to the animation controller used to playback that data.

The D3DX library provides a global function called D3DXCreateAnimationController for creating animation controllers. This function takes several parameters which define the limitations of the newly created animation controller. The function is shown below, followed by a discussion of its parameters.

```
HRESULT D3DXCreateAnimationController
(
      UINT MaxNumAnimationOutputs,
      UINT MaxNumAnimationSets,
      UINT MaxNumTracks,
      UINT MaxNumEvents,
      LPD3DXANIMATIONCONTROLLER *ppAnimController
);
```

UINT MaxNumAnimationOutputs

This parameter represents the maximum number of hierarchy frame matrices that the animation controller can manipulate during a single `AdvanceTime` call. By default, when our animation has been loaded via `D3DXLoadFrameHierarchyFromX`, this value will be set to the number of frames in the hierarchy that will be animated by the data stored in the file, including any child frames they may own. D3DX will register all of these matrices with the animation controller on our behalf when using D3DXLoadMeshHierarchyFromX.

To register additional matrices with the animation controller returned from D3DXLoadMeshHierarchyFromX we will need to clone the animation controller and adjust the MaxNumAnimationOutputs setting to provide space for those additional matrices. Individual matrix registration can be accomplished via the ID3DXAnimationController::RegisterAnimationOutput function, which we will study in a short while. This function will not generally be of concern to us when using D3DXLoadMeshHierarchyFromX, but it is needed for manual controller creation.

UINT MaxNumAnimationSets

This parameter describes the maximum number of animation sets that can be managed (registered via its RegisterAnimationSet method) at any one time by the animation controller. When loading the animation from an X file using D3DXLoadMeshHierarchyFromX, the default value is the total number of animation sets actually stored in the file. To register additional animation sets with the animation controller, we must clone a new one and update this value according to our needs. As mentioned earlier, many of the popular X file exporters currently support exporting only a single animation set. Thus the animation controller created will only provide storage for a single animation set to be registered. We can determine how many sets are registered with the controller using its GetMaxNumAnimationSets method.

UINT MaxNumTracks

This is the maximum number of mixer tracks that the controller should simultaneously support. Essentially it represents the maximum number of animation sets that can be simultaneously assigned to mixer tracks for blending. By default, this value equals 2 when we load animation data from an X file via D3DXLoadMeshHierarchyFromX. If we need additional tracks, we will simply clone a new controller to support however many animation mixer tracks are required.

Note that when we clone a new controller to add tracks, we can specify as many new tracks as we want, but each track will occupy roughly 56 bytes of memory (before data is assigned to it). Again, the

number of available tracks determines the maximum number of animation sets that can be simultaneously blended. Of course, we may have many more registered animation sets than available tracks on our animation mixer. For example, we might have 6 animation sets that should be played back (and blended) in groups of two, such that sets 1 and 2 might be played together, sets 3 and 4 might be played together and sets 5 and 6 might be played together. In this situation we would want to create the animation controller with a 'MaxNumTracks' of 2 but a 'MaxNumAnimationSets' of 6 since we need to manage six different animation sets but only need to playback two at a time.

UINT MaxNumEvents
This parameter allows us to specify the maximum number of key events that can be registered with the animation controller's sequencer. The animation controller sequencer is analogous to a MIDI sequencer used for music composition. Those of you that have worked with MIDI sequencer applications should recall that the sequencer allows for MIDI events to be placed on a given MIDI channel. These MIDI events might include instructions to change the keyboard instrument being used to play that track at a certain time or change the volume of a certain instrument at a certain time. With regards to animation sequencing, this feature affords us the ability to schedule special events to be triggered at certain points in the global timeline of the animation controller. These events include changing the playback speed of a specified track, modifying track blend weights, or jumping to/from locations on a track's local timeline at a specified global time. We will cover the sequencing functions a little later in the chapter.

The default animation controller created by D3DXLoadMeshHierarchyFromX is capable of storing a maximum of 30 key events. If this is not suitable, then we can clone the animation controller to extend the sequencing capabilities.

LPD3DXANIMATIONCONTROLLER *ppAnimController
The final parameter to this function is the address of a pointer to an ID3DXAnimationController interface. On successful function return, it will point to a valid interface for a D3DXAnimationController object.

10.7.2 ID3DXAnimation Controller Setup

The animation controller returned via the manual creation method discussed in the last section is initially empty. The first thing we will want to do is register all of the matrices we need the animation controller to animate, along with any animation sets we have created. While this will not be the case when using D3DXLoadMeshHierarchyFromX, we will need to understand the process when managing our own custom data.

Registering Frame Matrices

While it would not be difficult to write code that recursively steps through the hierarchy and registers any animated frame matrices, D3DX exposes a global function that will do this for us. It is called D3DXFrameRegisterNamedMatrices and is shown below along with a description of its parameters. We will pass this function a pointer to a D3DXFRAME structure from our hierarchy and it will register that

frame's matrix, along with any child frame matrices, with the animation controller passed in as the second parameter. Usually we will pass in a pointer to the hierarchy root frame so that all frame matrices (that have names) are registered with the animation controller.

```
HRESULT D3DXFrameRegisterNamedMatrices
(
    LPD3DXFRAME pFrameRoot,
    LPD3DXANIMATIONCONTROLLER pAnimController
);
```

LPD3DXFRAME pFrameRoot
This is the top level frame where hierarchy traversal will start. The function will register this frame's matrix (if it is named) and subsequently traverse its children, registering any named matrices it encounters. Passing the hierarchy root frame will ensure that all frame matrices that can be animated in the hierarchy are properly registered with the animation controller.

LPD3DXANIMATIONCONTROLLER pAnimController
The second parameter is a pointer to the animation controller that we would like the hierarchy matrices registered with.

While this function makes our job easy, there is one small issue which we must resolve. Recall that during animation controller creation, we will need to specify the maximum number of matrices that the controller will manage (MaxNumAnimationOutputs). As a result, we will need to know the count of named frame matrices in our hierarchy *before* the controller is created. While we could write our own traversal counting routine, D3DX again steps up to make our job easier by providing a global function called D3DXFrameNumNamedMatrices.

```
UINT D3DXFrameNumNamedMatrices
(
     LPD3DXFRAME pFrameRoot
);
```

The function takes a single parameter -- a pointer to a frame in our hierarchy (usually the root frame). It will return the number of named frames (including the passed frame) that exist in the subtree, including all named children of the passed frame. We can use the value returned to create the animation controller and then register the matrices using the previously discussed function.

Example

The following code snippet creates an animation controller and registers the matrices of the hierarchy. In this example it is assumed that pRoot is a D3DXFRAME pointer to the root frame of an already created hierarchy.

```
ID3DXAnimationController * pAnimController = NULL;
UINT MaxNumMatrices = D3DXFrameNumNamedMatrices ( pRoot );
D3DXCreateAnimationController ( MaxNumMatrices , 4 , 2 , 10 , &pAnimController );
D3DXFrameRegisterNamedMatrices ( pRoot , pAnimController );
```

The example builds an animation controller that is capable of managing four animation sets. It has an animation mixer with two channels (which means it is capable of blending only two animation sets at once) and can have ten key events set in its internal sequencer.

> **Note:** We remind you once again that these steps are only required when we manually create an animation controller. D3DXLoadMeshHierarchyFromX does all of this work on our behalf before returning a fully functional animation controller interface.

Registering Animation Sets

Once we have created our animation controller and registered all matrices that will be animated by our animation sets, it is time to register those animation sets with the animation controller. Until we do, the animation controller has no animation data (it has is a list of matrices, but no idea about how to fill them with animation results).

For each animation set that needs to be registered with the animation controller we call the ID3DXAnimationController::RegisterAnimationSet method. It accepts a single parameter -- a pointer to a valid ID3DXAnimationSet interface. By 'valid' we mean that it should already contain all of its keyframe data. This function would be called once for each animation set to be registered with the animation controller.

```
HRESULT RegisterAnimationSet( LPD3DXANIMATIONSET pAnimSet );
```

> **Note:** When an animation set is being registered, the controller calls the animation set's GetAnimationIndexByName function for each frame. Its goal is to retrieve either a valid index via the pIndex output parameter and D3D_OK, or D3DERR_NOTFOUND via the function result. Caching the animation indices bypasses the need to look up the index on the fly. It also tells the controller exactly which frames even have any animation data (those which resulted in D3DERR_NOTFOUND do not have a matching animation). Once this process is completed, AddRef is called to indicate the controller's use of the animation set that has just been registered.

It is important to realize that the number of animation sets we register with the controller must not be greater than the MaxNumAnimationSets parameter passed into the D3DXCreateAnimationController function. If you did not create the animation controller yourself and would like to know the maximum number of animation sets that can be registered with the animation controller, you can use the ID3DXAnimationController::GetMaxNumAnimationSets method of the interface as shown below. The function returns the total number of animation sets that can be registered with the animation controller simultaneously.

```
UINT GetMaxNumAnimationSets(VOID);
```

Assuming that we have previously created four animation sets and wish to register them with our animation controller, the code might look something along the lines of the following. The next code snippet assumes that pRoot points to a valid D3DXFRAME structure, which is the root frame of our hierarchy. It also assumes the four animation sets (and their interpolators) have already been created.

```
ID3DXAnimationController *pAnimController = NULL;
ID3DXAnimationSet *pAnimSet1= ' Valid Animation Set'
ID3DXAnimationSet *pAnimSet2= ' Valid Animation Set'
ID3DXAnimationSet *pAnimSet3= ' Valid Animation Set'
ID3DXAnimationSet *pAnimSet4= ' Valid Animation Set'

// Create animation controller
UINT MaxNumMatrices = D3DXFrameNumNamedMatrices ( pRoot );
D3DXCreateAnimationController ( MaxNumMatrices , 4 , 2 , 10 , &pAnimController );

// Register Matrices
D3DXFrameRegisterNamedMatrices ( pRoot , pAnimController );

// Register Animation Sets
pAnimController->RegisterAnimationSet ( pAnimSet1 );
pAnimController->RegisterAnimationSet ( pAnimSet2 );
pAnimController->RegisterAnimationSet ( pAnimSet3 );
pAnimController->RegisterAnimationSet ( pAnimSet4 );
```

At this point the animation controller will have all of the data (e.g. keyframe data and the matrices they manipulate) it needs to animate the hierarchy. However, no animation sets have yet been assigned to tracks on the mixer. This last step is something that will need to be done before we call AdvanceTime. We will discuss mixer track assignment in a moment.

Our application can retrieve the current number of animation sets that have been registered with the controller using the ID3DXAnimationController::GetNumAnimationSets method. This allows us to query how many of the animation set slots allocated for the controller are already occupied with animation sets. We can only register another animation set with the controller if:

```
pController->GetMaxNumAnimationSets() > pController->GetNumAnimationSets()
```

The function that retrieves the current total of registered animation sets is:

```
UINT GetNumAnimationSets(VOID);
```

Unregistering an Animation Set

Just as we can register animation sets with the animation controller, we can also unregister them. This might be useful if you have decided that you are finished with an animation set and will not need it again in the future. In this case you can unregister the set and then release its interface. It is also useful when you have more animation sets than your current animation controller can handle. Let us say for example that you had 10 animation sets which you planned on using but for some reason your animation controller was created to support only five registered sets at any one time. In this case you could register animation sets when they are needed and then unregister them when they are no longer needed to make room for other animation sets. The function is shown below and takes a single parameter -- a pointer to the ID3DXAnimationSet you wish to unregister.

```
HRESULT UnregisterAnimationSet(LPD3DXANIMATIONSET pAnimSet);
```

Note that to remove the animation set we must pass in the interface pointer. This function will call Release on the animation set when it relinquishes control (assuming that it owns the animation set we requested to be unregistered). So if we wanted to unregister an already existing animation set being managed by the controller that we do not currently have a pointer to, we would need to get access to it first. The following code uses the GetAnimationSet method of the ID3DXAnimationController interface (which we have not covered yet) to retrieve the ID3DXAnimationSet assigned to slot 3 in the animation controller's animation set array. We then unregister the animation set and finally release it.

```
LPD3DXANIMATIONSET pAnimSet;

// Fetch interface to 4th registered animation set
m_pAnimController->GetAnimationSet( 3, &pAnimSet );

// At this point, the animation set 'pAnimSet',
// will have a reference count of '2'.
m_pAnimController->UnregisterAnimationSet( pAnimSet );

// Release has been called, reference count is now '1'
pAnimSet->Release(); // Animation set deleted.
```

In the above code you can see that we retrieved the fourth animation set being managed by the controller (the one at index 3). Now that we have unregistered it, passing 3 to GetAnimationSet on a subsequent call will return a pointer to the animation set which used to exist at index 4. In other words, unregistering an animation set will shuffle all of the other animation sets that follow it back one slot in the animation controller's internal animation set array. It is also worth noting that any unregistered animation set will be removed from the mixer track if it is currently assigned to one. Any calls to GetTrackAnimationSet for the track which previously owned it will now return NULL until another animation set is assigned to that track.

When we call ID3DXAnimationController::Release, all of the registered animation sets are unregistered and their reference counts decremented. Therefore, we do not have to explicitly call UnregisterAnimationSet for each registered animation set before we release the animation controller interface.

> **Note:** When using D3DXLoadMeshHierarchyFromX to load animation data from an X file, all animation sets contained in the X file are created and registered with the animation controller on our behalf. There is no need to call the above function unless you need to register additional animation sets (not contained in the X file) with the animation controller. In that case you will probably need to clone the animation controller to make room for your extra animation sets.

Registering Matrices

As discussed previously, D3DXFrameRegisterNamedMatrices does all of the hard work of registering the matrices in a given hierarchy with an animation controller. This function is basically just traversing the hierarchy that we pass it and calling the ID3DXAnimationController::RegisterAnimationOutput method of the animation controller interface for each named matrix that it finds. If you need to manually

register some or all of your matrices with the animation controller, you can do so yourself using this method.

```
HRESULT RegisterAnimationOutput
(
    LPCSTR Name,
    D3DXMATRIX *pMatrix,
    D3DXVECTOR3 *pScale,
    D3DXQUATERNION *pRotation,
    D3DXVECTOR3 *pTranslation
);
```

LPCSTR Name
The first parameter is the name that you would like the output to be registered under. This name should match the name of an animation in an animation set.

D3DXMATRIX *pMatrix
The matrix pointer passed into this function is stored by the animation controller so that it can update the frame matrix with SRT data without traversing the hierarchy each time. When we use the D3DXLoadMeshHierarchyFromX function to load an X file which has animation data, this function will be called multiple times to store the matrix of each named frame in the hierarchy in the controller's matrix pointer table. While this may sound complicated, bear in mind that all we are doing is informing the controller about a matrix (usually the matrix of a D3DXFRAME structure) that is to receive the SRT data from animations with the passed name. This allows the controller to build a matrix pointer array for quick updates of the frame hierarchy matrices.

D3DXVECTOR3 *pScale
D3DXQUATERNION *pRotation
D3DXVECTOR3 *pTranslation
This function was updated in the 2003 summer update of the SDK and now NULL can be passed as the matrix parameter and the SRT data stored as separate components. This allows an application to store the SRT data from a given animation in either a matrix, or have it stored in a separate scale vector, a rotation quaternion and a translation vector instead. While an automatically created animation controller (one created by D3DXLoadMeshHierarchyFromX) will have this function called for each named frame with the final three parameters set to NULL, and will only register the frame matrix as an animation output, this does provide applications with the flexibility to store the SRT data output from a given animation in separate variables. For example, an application performing its own animation blending that still wishes to use the animation controller for SRT generation (keyframe interpolation) might decide not to have the output stored in a matrix but as a separate scale, rotation and translation that it can later blend into a matrix using its own custom techniques. If any parameter (except the first) is set to NULL, the associated SRT data will not be output. For example, if the pMatrix, pScale and pTranslation parameters are set to NULL and only a rotation quaternion pointer is registered, then only the rotation portion of the SRT data will be output. When using the full D3DX animation system, it is not likely that you will ever want the pMatrix parameter set to NULL, as this is usually a pointer to the matrix of a frame in the hierarchy that you wish to be updated when AdvanceTime is called. It is often the case that both the matrix will be registered as an animation output along with some or all of the final three parameters. This allows the animation controller to update the frame in the hierarchy (pMatrix) and also return SRT data to the application, which may be used to calculate physics for example.

It is important that you do not register more outputs with the animation controller than the maximum number allowed. The maximum number of output that can be registered with the controller is set when the animation controller is created (the MaxNumAnimationOutputs parameter) using the D3DXCreateAnimationController function. You can retrieve the maximum number of supported outputs by calling the ID3DXAnimationController::GetMaxNumAnimationOutputs function.

```
UINT GetMaxNumAnimationOutputs(VOID);
```

This function returns an unsigned integer value describing the maximum number of outputs that can be registered with the animation controller at any one time. As a result, this value also tells us the maximum number of frame matrices that can be simultaneously animated by the controller. A single output can be a combination of a matrix, a scale and translation vector and a rotation quaternion.

We have now covered how to create and configure the animation controller, and how to register all required matrices and animation sets. So the animation controller now has all of the data it needs for animating our hierarchy matrices. Again, we will rarely need to perform any of these tasks if we are simply loading animated X files, because D3DXLoadMeshHierarchyFromX will do the work for us.

The next step is to understand how to use the methods of the ID3DXAnimationController interface to configure the animation mixer. This will allow us to start experimenting with how our animation sets are played back in our application.

10.8 The Animation Mixer

Although we have registered our animation sets with the animation controller (or it has been done for us by D3DXLoadMeshHierarchyFromX), we are not quite ready yet to start playing them back. Earlier we learned that we will use the ID3DXAnimationController::AdvanceTime method to progress along its global timeline and animate the hierarchy. While this is certainly the case, keep in mind that we might have multiple animation sets registered with the animation controller and that we will want the ability to play back only a subset of those animation sets at any given time. For example, if we had an animated character hierarchy with three animation sets (‘Walk’, ‘Swim’, ‘Die’), we will probably want to play back only one of these sets at any given time.

The animation mixer allows us to select the animation set(s) we wish to use for animating the hierarchy each time AdvanceTime is called. It also allows us to have multiple animation sets assigned to different tracks so that they can be blended together to produce more complex animations.

When the animation controller is first created, it is at this time that we determine the maximum number of mixing tracks that the animation mixer will be able to use. This lets us scale the animation mixer to suit our needs so that memory is not wasted by maintaining tracks that our application will never use. You should recall from our discussion of the D3DXCreateAnimationController function that the MaxNumTracks parameter defines the maximum number of tracks for the animation controller. To use the analogy of an audio multi-track mixing desk once again, these mixing desks come in different shapes and sizes. The more tracks an audio mixing desk has, the more individual sounds we can simultaneously mix together. For example, guitar, bass, drums, and vocals are generally assigned their own separate tracks when your band is recording an album. This allows the studio sound engineer to isolate and tweak each individual instrument sound for better results (or to add effects, etc.). Although each track maintains only the data for that particular instrument, all tracks will be played back simultaneously on the final album, blended together to produce the song in its complete form.

With the animation mixer, the number of tracks defines the maximum number of animations sets that can be blended together at any one time. So in order to play an animation set, we must first assign it to a track on the animation mixer and then enable that track. Tracks can be enabled and disabled by our application at will, so we have total control over the process in that respect. Next we will discuss how to assign animation sets that have been registered with the animation controller to tracks on the animation mixer. This way they will be ready for use the next time our application calls the ID3DXAnimationController::AdvanceTime function.

10.8.1 Assigning Animation Sets to Mixer Tracks

When an animation controller object is *manually* created and its animation sets are registered, by default none of these animation sets will be assigned to any tracks on the animation mixer. Before we play an animation we must select the appropriate animation set(s) and assign them to the mixer track(s) we wish to use. This is analogous to the painter who has many jars of colored paints in his workshop, but will only put a few blobs of paint on his palette to use in the current painting. We can think of the registered animation sets as the available jars of paint. Some of these sets will be placed into the mixer (the palette) to be used to create the final animation (the painting). Note that the artist may decide to smear some of the paint blobs together on his palette to create new color blends for the painting. We can do the same thing with our animation mixer.

Assuming that we have registered our animation sets with the animation controller, we can assign any animation set to a valid track on the animation mixer. By a ‘valid’ track, we simply mean a track with an index in the range [0, MaxNumTracks - 1]. Unlike the DirectX lighting module, where we can assign lights to any arbitrary lighting slot index just so long as we do not enable too many lights simultaneously, animation mixer tracks are always zero-based and consecutive in their numbering scheme. Therefore, if we created an animation controller that supports a maximum of eight mixer tracks, this means that we can assign a different (registered) animation set to tracks 0 through 7 on the

animation mixer. Note that the index of an animation set is not related to the index of the track to which it can be assigned. The index of an animation set is merely representative of the order in which it was registered with the controller. It is perfectly acceptable to assign animation set 5 to animation mixer track 2 for example.

Keep in mind that when the animation controller is created during the D3DXLoadMeshHierarchyFromX call, all animation sets will be registered, but only the last animation set defined in the file will be assigned to a track (track 0, specifically). By default, the animation controller will support a maximum of two tracks and the second track will be disabled with no assigned animation set. If you require additional tracks to produce your animations, you must clone the animation controller to extend its capabilities.

We assign an animation set to a mixer track on the animation mixer using the ID3DXAnimationController::SetTrackAnimationSet function. The first parameter is the zero-based number of the track we want the animation set assigned to. The second parameter is the interface pointer of the animation set being assigned.

```
HRESULT SetTrackAnimationSet( DWORD Track,
                              LPD3DXANIMATIONSET pAnimSet );
```

The animation set passed in must already be registered with the controller. Any previous animation set already assigned to the specified track will be unassigned before the new animation set assignment takes place.

If you do not have the actual interface of the animation set handy, but you know the index of the animation set (based on when it was registered) then you can use the ID3DXAnimationController::GetAnimationSet function to retrieve its interface.

```
HRESULT GetAnimationSet( DWORD iAnimationSet,
                         LPD3DXANIMATIONSET *ppAnimSet );
```

We pass this function the index of the registered animation set interface we want to retrieve and the address of an animation set interface pointer which will store the returned result. This function will increment the reference count of the object before returning the interface, so be sure to release it when you have finished with it.

Next we see some example code that acquires an interface pointer for the third registered animation set (AnimationSet[2]) and assigns it to track 7 on the animation mixer. This code assumes that the animation controller was created with a minimum of an 8-track animation mixer.

```
// Set up track 0 for the animation mixer
LPD3DXANIMATIONSET pAnimSet;
m_pAnimController->GetAnimationSet( 2, &pAnimSet );
m_pAnimController->SetTrackAnimationSet( 7, pAnimSet );
pAnimSet->Release();
```

It is much more intuitive to find animation sets by name (ex. “Jump”) and as we might expect, we can also retrieve the animation set interface by name also. This function, which is shown next, accepts a string containing the name of the animation we are searching for and the address of a pointer to an ID3DXAnimationSet interface. On successful function return this will point to a valid animation set.

```
HRESULT GetAnimationSetByName( LPCSTR pName,
                               LPD3DXANIMATIONSET *ppAnimSet);
```

In the following example, we show how we might assign an animation set called “Shoot” to mixer track 3 (i.e., the fourth track).

```
// Set up track 0 for the animation mixer
LPD3DXANIMATIONSET pAnimSet;
m_pAnimController->GetAnimationSetByName( “Shoot”, &pAnimSet );
m_pAnimController->SetTrackAnimationSet( 3, pAnimSet );
pAnimSet->Release();
```

So as you can see, we have two useful ways to retrieve the interface for a registered animation set that we wish to assign to a track. We can retrieve its interface by name or by index.

```
HRESULT GetTrackAnimationSet( DWORD Track,
                              LPD3DXANIMATIONSET *ppAnimSet );
```

ID3DXAnimationController exposes another animation set retrieval function called GetTrackAnimationSet which allows us to specify a track number and get the interface for the animation set currently assigned to that track. This is handy if we wish to fetch the animation set assigned to a given track on the mixer and assign it to another track. It is also useful if we want to unregister the animation set because we have no further use for it. Note that unregistering an animation set will also remove its track assignment.

The function takes two parameters -- the first is the index of the mixer track containing the animation set we wish to obtain an interface for, and the second is the address of an ID3DXAnimationSet interface pointer. The latter parameter will point to a valid interface to the animation set on function return. If no animation set is assigned to the specified track, the interface pointer will be set to NULL.

Once again, we must remember to release the interface when we are finished with it so that the underlying COM object will be properly cleaned up when it is no longer needed by any modules.

The following code shows how we might assign three animation sets (sets 2, 5, and 8) to animation mixer tracks 0, 1, and 2. It assumes that the animation controller has already been created with a minimum of 9 pre-registered animation sets and at least a 3-track mixer. This example will be typical of code that might be executed in response to a certain action or event taking place in our game logic. For example, imagine that this particular animation controller is managing the hierarchy of an animated character and that the animation sets are mapped as follows:

Set 2 = Walk
Set 5 = Fire Weapon
Set 8 = Act Wounded

If the character was currently experiencing all of these states then we could assign them to the first three tracks of the animation mixer. Then the next time we call AdvanceTime, the character will be animated using the blended result of these three animations sets. As a result the character would fire his weapon while walking along, behaving like he has sustained injuries. This next example will assume that the PlayerAction variable is a DWORD that has bits set that our application identifies with the walking, firing, and wounded action states.

```
if ((PlayerAction & WALK) && (PlayerAction & FIRING) && (PlayerAction & WOUNDED) )
{
      LPD3DXANIMATIONSET pAnimSet2 = NULL;
      LPD3DXANIMATIONSET pAnimSet5 = NULL;
      LPD3DXANIMATIONSET pAnimSet8 = NULL;

      m_pAnimController->GetAnimationSet( 2, &pAnimSet2 );
      m_pAnimController->GetAnimationSet( 5, &pAnimSet5);
      m_pAnimController->GetAnimationSet( 8, &pAnimSet8);

      m_pAnimController->SetTrackAnimationSet( 0, pAnimSet2 );
      m_pAnimController->SetTrackAnimationSet( 1, pAnimSet5 );
      m_pAnimController->SetTrackAnimationSet( 2, pAnimSet8 );

      pAnimSet2->Release();
      pAnimSet5->Release();
      pAnimSet8->Release();
}
```

The code block above sets the animations we want to play on this animation controller (at least until the character enters another action mode). When we call ID3DXAnimationController::AdvanceTime in our render loop, these animations can then be blended together for our final on-screen result. However, this is not all we will usually have to do. After all, we need to find the right balance between these three animations to make sure that the results are correct. Mixing animations is more an art than a science, and we will often have to experiment a bit to find good blends that work for us. Fortunately, each track on the animation mixer has several properties that can be set to control how the animation applied to that track is played out and how that track is blended with animation sets assigned to other tracks.

In the next section we will examine how to set up the individual tracks of the animation mixer so that they play their assigned animations in the desired way.

10.8.2 Animation Mixer Track Configuration

Every animation mixer track has five configurable properties which determine the behavior of the track and thus the animation set assigned to it.

The first property **enables/disables** the track. If a track is disabled it will not be used in the AdvanceTime call regardless of whether it contains an animation set. Being able to enable/disable tracks on the fly is a useful feature in a game. Imagine that we have two animation sets assigned to our mixer: track one stores a ‘Walk’ animation which we repeatedly loop as the character navigates the world. On

the second track we store a short animation called 'Wave' which makes the character wave his hand to other characters in the world in greeting as he passes them by. Since we would only want the second animation to play when the character spots a friend, we would disable this track by default. At this point, every call to AdvanceTime would only update the hierarchy using the 'Walk' animation set assigned to track one. But once the character spots a buddy, we could enable track two for a set period of time (a few seconds perhaps) before disabling it again. In the few seconds that track two is enabled, any calls to AdvanceTime would blend the two animation sets together such that the character model would be walking and waving at the same time.

The next important mixer track property is **weight**. Going back to our audio mixing desk analogy, we can think of the weight of a track as being akin to the volume knob. Increasing the volume of a track will make the instrument assigned to that track louder in the final mix. The weight property of the animation track is similarly a way to scale the influence of its assigned animation set on the hierarchy. The track weighting value is usually in the range [0.0, 1.0] where a value of 1.0 means the animation set assigned to that track manipulates the hierarchy at its full strength. A value of 0.0 means the animation set will not influence the hierarchy at all. This value can be set higher than 1.0 although the resulting hierarchy transformation might not be what you expect. We will look at a track's primary weight value in more detail in a moment.

The third configurable property of a track is its **speed**. Speed is a scalar value which allows us to define how fast we want the animation set to be played back. A speed value of 1.0 means 'normal speed' (i.e. the speed that it would play at by default as intended by the animation creator). Specifying a value of 2.0 for instance means the animation set assigned to the track would play out twice as fast. This might sound like a strange feature to have, but it is actually nice to have the flexibility to play the animation sets back at different speeds (imagine that you wanted to do a slow-motion sequence where all animations ran at half time). Setting the speed of a track is another feature that allows us to decouple the global time of the controller from the timer of a given track. We can think of the speed parameter of a track as being a scalar that is multiplied by the elapsed time passed into the Advance Time call prior to being added to the track timer. As an example, let us imagine that we pass into AdvanceTime an elapsed time of 2 seconds. We know that the global timer of the controller would have two seconds added to it. However, before this elapsed time is also added to the timer of each track, it is scaled by the speed parameter of that track. If the speed of the track was set to 0.25, then the actual elapsed time added to the track would be ElapsedTime*0.25 = 0.5. As you can see, although the global time and the rest of the animation tracks have moved along their time lines by 2 seconds, the track in question has only had its timer updated by ½ a second. Therefore, the animation set assigned to this track appears to be playing back in slow motion compared to other sets being played. In this example, it is playing back at ¼ of the speed of the global timer. Obviously, speed values greater than 1 (the default) will make the animation set play back faster than global time. For example, a speed value of 2 will cause the animation to play back at twice its speed.

The fourth property of a mixer track is a time value called ***position***, which is essentially the value of the timer for that track. Therefore, in DirectX terminology, the global time is the time of the animation controller which is the sum of total elapsed time, and the position is the current value of a single track timer (which itself is the sum of all elapsed time passed onto that track during updates). The ***periodic position*** is the value of the track ***position*** mapped into the period of the animation set, taking into account looping and ping-ponging. It is the track position that is passed into the

ID3DXAnimationSet::GetPeriodicPosition function to generate an animation set local time that is ultimately passed on to the ID3DXAnimationSet::GetSRT function.

To understand the three layers of time used by the D3DX animation system, below we show a simplified pseudo-code version of how the ID3DXAnimationController::AdvanceTime function would apply updates.

```
AdvanceTime(  ElapsedTime )
{
    // Add elapsed time to the animation controllers global time
    GlobalTime += ElapsedTime;

    // Loop though each enabled track
    For (Each Enabled Track)
    {
       // Update the position of the track timer scaling by track speed
       Track->Position += (ElapsedTime * Track->Speed)

       // Fetch animation set assigned to current track
       AnimationSet = Track->GetTrackAnimationSet( )

       // Map the track timer ( position ) into a periodic position for the set
       PeriodicPositon =AnimationSet->GetPeriodicPosition( Track->Position )

       // Use periodic position to generate SRT data for each animation in this set
        For ( Each Animation In Animation Set)
        {
              AnimationSet->GetSRT( PeriodicPosition , Animation ,
                                    &scale , &rot , &trans )

          .. Do other stuff with SRT data here …
        } // Each Animation

     } // End Enabled Track
}
```

We will usually want to set the position of a track to zero before the animation set is played for the first time, so that the animation set assigned to that track starts at the beginning.

While we will often have no need to interfere with the local time (the position) of a mixer track and will usually allow the animation controller to advance them on our behalf when we call AdvanceTime, we do have the ability to override this behavior and offset a track's animation set to a specific track local time. Imagine that we initially assigned two animation sets to tracks 1 and 2. We could set track 1's position to 0 and track 2's position to 5. When we start playing our animation using AdvanceTime, track 1 would start playing its animation set from the beginning while track 2 would start playing from 5 seconds in (assuming the period of the animation was at least 5 seconds long). This is probably a more useful feature if your animation set is actually a container filled with multiple animations (as is often the case for the default export of X files). In such a case your engine would need to know where particular animation routines started and ended in the set (ex. 'Run' animation from 0 seconds to 2 seconds, 'Shoot' animation from second 3 to second 6, etc.) and pass these times appropriately. It is a bit more complicated of course because your engine will also need to control when the animation should stop and

be reset to play again from the beginning (which is why splitting up animation sets according to individual animation routines makes life much easier – as we will do in Lab Project 10.2).

The fifth and final property that we can assign to a mixer track is a **priority** value. With it, each track can be assigned one of two possible priorities: high or low. Low priority tracks are actually blended together separately from high priority tracks. Later, the two blended results are blended together using the animation mixer's **priority blend weight** (which is another weight we can set). For example, assume that we have an 8-track mixer and we assign the first four tracks a low priority and the second four tracks a high priority. When we call AdvanceTime, the low priority tracks (0-3) would be blended together using their respective per-track weights and the high priority tracks (4-7) would be blended together using their respective per-track weights. Finally, the results of both blends for a given frame are blended together using the priority blend weight set for the mixer. This is a single property value for the entire animation mixer that we can configure as needed. If it was set to 0.25 for example, it means that after the low priority tracks have been blended (we will call this Result 1) and the high priority tracks had been blended (we will call this Result 2), the final transformations applied to the hierarchy would be a combination of 25% of Result1 and 75% of Result2. This provides another layer of animation blending control beyond the per-track blend weights. Figure 9.8 gives us a graphical idea of the concepts just discussed. It demonstrates an 8-track mixer with all of the configurable properties outlined previously, along with the input (animation sets) and output (transformations).

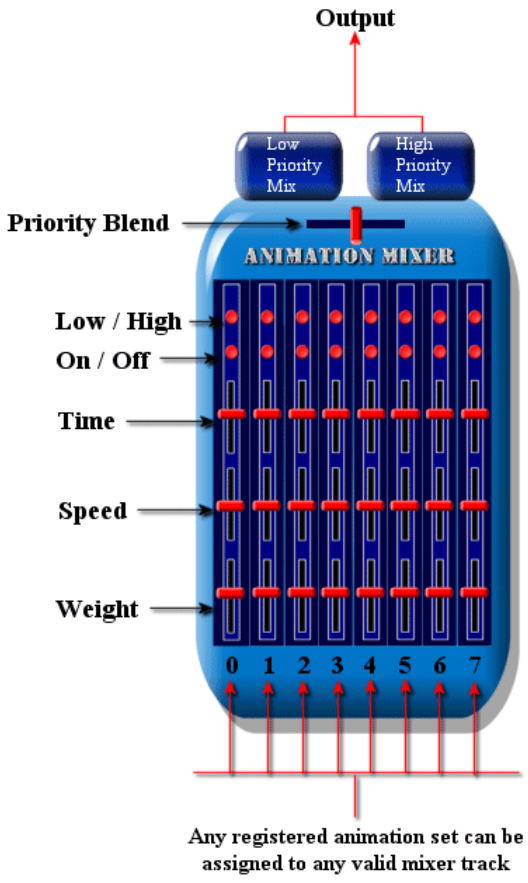


Figure 9.8

Enabling / Disabling a Track

The first thing we will usually want to do when we assign an animation set to a track is make sure that track is enabled. This will ensure that the track and its assigned set are used in the next call to AdvanceTime. We enable and disable tracks using the ID3DXAnimationController::SetTrackEnable method.

```
HRESULT SetTrackEnable( UINT Track, BOOL Enable );
```

The first parameter to this function is the index of the mixer track we wish to enable or disable. The second parameter is either TRUE or FALSE depending on whether we would like to enable or disable the mixer track respectively. To mix a track with other tracks, this flag must be set to TRUE. Conversely, setting this flag to FALSE will prevent the specified track from being mixed with other tracks. You should not leave tracks enabled if they have no animation sets assigned to them.

Setting Track Weight

Setting the weight of a track controls the influence of the animation set assigned to it (i.e. how strongly it manipulates the hierarchy). We can set the per-track weight value using the ID3DXAnimationController::SetTrackWeight function.

```
HRESULT SetTrackWeight( UINT Track, FLOAT Weight );
```

The first parameter is the index of the track we are specifying the weight for and the second parameter is the floating point weight value itself. This is the primary track weighting value and it will generally be specified in the [0.0, 1.0] range. Although you can specify values outside of this range if it serves your purpose, there is the potential for some strange results. All weights for all active tracks are essentially normalized anyway so if you have only one track and set its weight to 0.5, it will essentially have a weight of 1.0 anyway. The controller makes sure that the sum of all track weights equals one prior to the blending process.

Let us imagine that we have a single cube in our scene hierarchy, and that we have two separate animation sets that will manipulate that cube. The first animation set translates the cube 100 units to the right and the second set translates the cube 100 units to the left. If both of these sets were placed in separate tracks, and both had a weight of 1.0, nothing would actually happen to the cube. Each set would produce translations which counteract one another:

```
(100 * 1.0) + (-100 * 1.0) = 0
```

However if we were to use a weight of 0.4 for the first set (which moves the cube 100 units to the right) the cube itself will have moved 60 units to the left by the end of the animation:

```
(100 * 0.4) + (-100 * 1.0) = -60
```

While this example is very simple, you can certainly understand the basic principles involved here. This is a generic blending formula that we have seen many times before (in Chapter Seven for example). In a game, we might use the weight value for blending an animation set of a character firing a gun (with only the associated animations applied to the arms), with a recoil animation set that we can blend at different strengths based on the type of weapon being fired.

It is worth noting that, by definition, the weight value can be used to define the strength of an animation, even in cases where hierarchy frames are not being manipulated by more than one track. Although keep in mind that it will have no effect on the animation. For example, if we had two tracks and neither manipulated the same hierarchy frames as the other one, the weight would not scale the strength of the animation applied to the respective matrices. You will probably set your per-track weights to 1.0 most of the time so that each animation set affects the hierarchy in the exact way intended by the creator of the animation data. This is certainly true if your animation sets generally influence their own unique portions of the hierarchy.

As discussed previously, the per-track weights are blended with other tracks assigned to the same priority group (high or low). The results of each priority group are then blended together using the priority blend weight of the animation controller to produce the final SRT data for a given frame. Therefore, even if a track is assigned a weight of 0.5, it does not necessarily mean that it will influence the hierarchy at half strength. The final strength of influence may be reduced or boosted depending on how the two priority groups are combined in the final weighting stage.

Setting Track Speed

Recall that while we pass an elapsed time (in seconds) to the controller's AdvanceTime method to update the global timer of the controller, each track maintains it own local timer which is also updated with the elapsed time passed in. The current track time is referred to as the track's position. For each track we can set a speed value (a single scalar) that is essentially used to scale the elapsed time we pass in and thus the amount that we wish to increment the timer of each track. If this scalar is set to 1.0 (the default) then the animation set will play back at its default speed -- the speed intended by the animation creator -- and the elapsed time passed into the AdvanceTime function will be added directly to the timer of the track without modification. Alternatively, if we set this value to 0.5, then the animation set assigned to that track would play out at only half the speed. This is because the elapsed time is scaled by the track speed before being added to the track's timer. A track with a speed setting of 0.5 will progress at half the speed of the global timeline of the controller. If a track is initially set with a speed value of 0.25, then at twenty seconds of global time, the track's timer will be at 5 seconds. As it is the track's timer that is mapped to a periodic position and used to generate SRT data, this directly effects the speed at which the animation set assigned to the track plays out.

```
HRESULT SetTrackSpeed( UINT Track, FLOAT Speed );
```

The first parameter should be the index of the track you wish to set the speed for. The second parameter is a floating point value indicating the desired speed scale value.

Setting Track Animation Position

As discussed earlier, each track maintains its own local timer. Every time the AdvanceTime method is called and the global timer is updated, so too are the timers for each track. The elapsed time passed in will be scaled by the speed of a track before being used to increment the track timeline. The current value of a track's timer is referred to as its position.

Not only do we have the ability to control the speed at which a track's counter updates using the elapsed time and its speed, we also have the ability to set the track timer to any arbitrary position. This allows us to cause the animation set to jump to a specific local time at a specific global time.

Usually, we will set the initial position for each track to zero before we play any animations, to ensure that all of the animation sets start from the beginning. Nevertheless, we can call the ID3DXAnimationController::SetTrackPosition function whenever we wish to set (or offset) the local timer of a specific track to a specific track position. We could, for example, have an animation set that is looping but needs to always restart from the beginning every time an event happens in the game, regardless of its current periodic position.

```
HRESULT SetTrackPosition( UINT Track, DOUBLE Position );
```

The first parameter is the track we wish to set the time for and the second parameter is the position we wish to set the track to. The next time we call the AdvanceTime method, all additions to the timeline will be relative to this new track position. As it is the track time that is mapped into a periodic position in the animation set, setting the position of a track directly effects the periodic position also.

The next code snippet demonstrates how we might want to set position of the mixer to start from the beginning (regardless of where this track may currently be in its timeline) in response to the 'FireWeapon' command. This will cause the animation set assigned to that track to restart regardless of whether it is configured to loop or not. It is assumed when looking at this code that the fTime parameter contains the elapsed time (in seconds) that has passed since the last frame. This is the value that will be added to the global time of the controller and used to increment the timer of each track (scaled by track speed). It is also assumed that there are multiple animation sets assigned to tracks on the animation mixer and that these animation sets will loop when they reach their end. When the fire button is pressed however, the animation set assigned to track 2 on the mixer restarts because we set the track time to zero, effectively forcing a restart of just that animation set. This will not affect any of the other animation sets assigned to other tracks; they will continue to play out as normal without any knowledge of the restart in track 2.

```
void UpdateAnimations ( double fTime , ID3DXAnimationController * pController)
{
      if (FireButton)
            pController->SetTrackPosition ( 2 , 0.0f );

      // Set the new  global time
      pController->AdvanceTime ( fTime );
}
```

It is worth remembering that although we set the track timer to zero, we then proceed to call the AdvanceTime method passing in the advanced time. Therefore, the next time the SRT data for this animation set if generated, the position of the track will in fact be 0 + ElapsedTime, as we would fully expect. This track time would be mapped to the periodic position of the animation set and used to fetch the two bounding keyframes used for interpolation.

> **Note:** Remember, the AdvanceTime method updates the matrices of the individual hierarchy frames. It is only when you traverse the hierarchy and combine the newly updated matrices that you will be ready to draw the final results.

Setting Track Priority

This function allows us to assign each track to one of two blending groups: a high priority group or a low priority group. Each group will have its tracks blended together separately using the per-track weight and eventually the blended results for each group will be blended into a final result used to update the hierarchy matrices. As we have the ability to alter the way in which the two priority blend groups are combined (using the controller Priority Blend Weight which we will discuss in a moment) this provides us a second layer of blend control.

This method is shown below and as you can see it takes two parameters. The first is the track index and the second is a member of the D3DXPRIORITY_TYPE enumeration.

```
HRESULT SetTrackPriority( UINT Track, D3DXPRIORITY_TYPE Priority);
```

The D3DXPRIORITY_TYPE enumeration is defined by D3DX as:

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

As you can see there are two choices (D3DXPRIORITY_LOW or D3DXPRIORITY_HIGH) indicating which of the two possible groups the track should be assigned to. Naming these groups High Priority and Low Priority is perhaps a little misleading as it incorrectly suggests that tracks assigned to the high priority groups will always have a higher influence on the hierarchy than those in the low priority groups. In actual fact, we will see in a moment how the animation controller lets us set a priority blend weight for the entire controller (a value between zero and one) which allows us to control how the high and low priority blend groups are blended. A value of 0.5 for example would blend both groups with equal weight. Alternatively, this value could also be set such that the low priority group has a greater influence. Rather than thinking of these groups as high and low priority, it is perhaps easier to think of them as being just two containers which can have arbitrary priority assigned to them. We will see how to set the priority blend weight of the animation controller in a moment.

Setting Track Descriptors

We have now seen how to set a track's speed, weight, priority group and position as well as how to enable or disable the track at will. The functions we have looked at are typically going to be called while our animation is playing so that we can adjust the track properties as needed, perhaps in response to some game state change. As it turns out, we can also setup these values (typically when we first create the animation controller) using a single function call. This function is called ID3DXAnimationController::SetTrackDesc and it allows us to set the speed, weight, priority groups and position of a track with a single call. The method is shown below.

```
HRESULT SetTrackDesc( UINT Track, D3DXTRACK_DESC *pDesc );
```

The first parameter is the number of the track we wish to set the properties for and the second parameter is the address of a D3DXTRACK_DESC structure containing the values for each property. The structure is shown below followed by a description of each member. The meaning of each member should be obvious to you given the functions we have covered in the previous section.

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

DWORD D3DXPRIORITY_TYPE
This parameter should be set to either `D3DXPRIORITY_LOW` or `D3DXPRIORITY_HIGH`. This allows us to assign the track to one of two blend groups: a high priority group or a low priority group as discussed earlier. Each group will have its tracks blended together separately and eventually the blended results for each group will be blended into a final result used to update the hierarchy matrices.

FLOAT Weight
This value has exactly the same meaning as the 'Weight' parameter we pass to the SetTrackWeight method. It allows us to set the current weight of the track which determines how much influence the assigned animation set has on the hierarchy.

FLOAT Speed
This member has exactly the same meaning as the 'Speed' parameter in the SetTrackSpeed function. It allows us to set a scalar value that is used by the controller to scale the animation time update for a given animation set. A value of 1.0 will run the animation at its default speed while a value of 2.0 will run the animation set at twice the default speed, and so on.

DOUBLE Position
This member has exactly the same meaning as the 'Position' parameter in the SetTrackPosition function. It allows us to set the timer of a track to a specific position without affecting any other tracks.

As a simple example, this allows us to restart the animation set assigned to a single track on the mixer without affecting other tracks.

BOOL Enable
This member is a Boolean variable where we pass in 'TRUE' or 'FALSE' to enable or disable the track, respectively.

This single function can be used instead of all the standalone functions we discussed in the previous sections. Whether you are initialising the properties of each track or simply changing the properties of a track midway through your game, this function provides a nice way of assigning multiple properties to a track with a single function call.

Finally, our application can retrieve the properties of a given track by using the ID3DXAnimationController::GetTrackDesc function.

```
HRESULT GetTrackDesc( DWORD Track, D3DXTRACK_DESC *pDesc );
```

The first parameter should be the track index we wish to retrieve the property information for and the second parameter should be the address of a D3DXTRACK_DESC structure to be filled with results.

Priority Blending

Each mixer track can be assigned to either a low priority group or a high priority group. The tracks of each group are blended together and then the resulting transformations for each group are blended together using what is referred to as the *priority blend weight* of the animation mixer. This is a global setting for the animation mixer and it controls how the low priority blend result is combined with the high priority blend result. We use the ID3DXAnimationController::SetPriorityBlend function to set this weight as a floating point value between 0.0 and 1.0.

```
HRESULT SetPriorityBlend ( FLOAT BlendWeight );
```

Tracks are assigned to priority groups using the either the SetTrackPriority function or the SetTrackDesc function discussed in the last section. Group assignment is based on one of the following flags:

D3DXPRIORITY_LOW – Track should be blended with all other low priority tracks ***before*** mixing with the high-priority result.

D3DXPRIORITY_HIGH – Track should be blended with all other high priority tracks, ***before*** mixing with the low-priority result.

Let us return to an earlier example to get some more insight into this concept. Recall that we have two tracks: one translates 100 units to the right and the other translates 100 units to the left. When each had a per-track weight of 1.0 we ended up with no movement because they cancelled one another out. However, if we were to specify them uniquely as low/high priority tracks, we see the following:

```
Track1: ( 100 * 1.0 ) = 100       // High priority track
Track2: ( -100 * 1.0 ) = -100     // Low priority track

// First Test
BlendWeight = 0.25 // One quarter high, three quarters low

Output = Track1 + ((Track2 - Track1) * BlendWeight)
Output = 100 + ((-100 - 100) * 0.25) = 50
Output = One quarter from 100 (high) and three quarters from -100 (low)

// Next Test
BlendWeight = 0.5       // Half way between low and high
Output = Track1 + ((Track2 - Track1 ) * BlendWeight)
Output = 100 + ((-100 - 100) * 0.5) = 0
Output = Half way between -100 (high) and 100 (low)

// Last Test
BlendWeight = 0.75      // Three quarters high, one quarter low

Output = Track1 + ((Track2 - Track1) * BlendWeight)
Output = 100 + ((-100 - 100) * 0.75) = -50
Output = Three quarters from 100 (high) and one quarter from -100 (low)
```

You can see that this is really just a simple linear interpolation between high and low priority tracks.

Our application can retrieve the current priority blend mode of the animation mixer by calling the ID3DXAnimationController::GetPriorityBlend function. It returns a single floating point value as a result.

```
FLOAT GetPriorityBlend( VOID );
```

10.8.3 Animation Mixer Configuration Summary

You should now be comfortable with the notion of assigning animation sets to tracks on the animation mixer and modifying track properties to effect the blended animations that are output. The following code snippet demonstrates how we might assign animation sets to two tracks in the animation controller and configure them in different ways. This code assumes that the animation controller has already had its animation sets registered.

```
pAnimController->SetTrackAnimationSet ( 0 , pAnimSetA );
pAnimController->SetTrackAnimationSet ( 1 , pAnimSetB );

D3DXTRACK_DESC Td;
Td.Priority =     D3DXPRIORITY_HIGH;      // Assign High Priority Group
Td.Enable   =     TRUE;                   // Enable Track
Td.Weight   =     1.0f;                   // Set Default Weight
Td.Speed    =     1.0f;                   // Set Default Speed
Td.AnimTime =     0.0f;                   // Start animation set at beginning
pAnimController->SetTrackDesc ( 0, &Td );
```

```
Td.Flags    =     D3DXPRIORITY_LOW ;      // Assign Low Priority
Td.Enable   =     TRUE;                   // Enable Track
Td.Weight   =     1.0f;                   // Set Default Weight
Td.Speed    =     0.5f;                   // Play track 1 at half its actual speed
Td.AnimTime =     12.0f;                  // Start this animation set 12 seconds in
pAnimController->SetTrackDesc ( 1, &Td );

// We have now set the track properties.
// Let us set the priority blend weight to distribute blending
// between high and low priority groups equally (a 50:50 blend)
pAnimController->SetPriorityBlend ( 0.5f );
```

Over time, you will find lots of creative ways to make use of animation mixing in your applications. Perhaps in one application you will find that you need only per-track weighting to achieve your desired results and you will forego using the priority blending system. In another application you may decide that you will mix two types of animation methods together (ex. keyframe animation with inverse kinematics) and find that the priority blending system works well for you. Either way, we have certainly seen thus far that D3DX provides quite a sophisticated system that, in the end, can save us a good deal of coding time.

10.8.4 Cloning the Animation Controller

Earlier in the chapter we learned that during the creation of our animation controller we will specify certain limitations such as the maximum number matrices or maximum number of animation sets that will be managed. While this presents no problem when we manually create an animation controller, because we can set the limits the animation controller to suit our needs, we might find ourselves disappointed with the default restrictions specified by D3DXLoadMeshHierarchyFromX when we load our animation data from an X file. The animation controller returned from D3DXLoadMeshHierarchyFromX has the following default limits:

MaxNumMatrices = Number of animated matrices in the X file hierarchy
MaxNumAnimSets = Number of animation sets in the X file (usually 1)
MaxNumTracks = 2
MaxNumEvents = 30

The one limitation that may be the hardest to accept is the two-track animation mixer which restricts us to blending only two simultaneous animations. It may also be problematic (as logical as the design decision was) to limit ourselves to the number of animation sets defined in the X file being the maximum number that can be registered. For example, although the X file might only contain a single animation set, we may want to generate another animation set programmatically and register that with the controller too. We would not be able to do so under the current circumstances, as this would exceed the maximum number of animation tracks that can be registered with the controller.

Fortunately, we are not locked in to the defaults that D3DXLoadMeshHierarchyFromX imposes. Just as we are able to clone a mesh to extend its capabilities (Chapter Eight), we can also clone an animation controller and modify its capabilities. Cloning creates a new animation controller with the desired

capabilities while preserving the animation, matrix, and animation set data of the original controller. That data will be copied into the newly generated controller such that it is ready to animate our hierarchy immediately. Once we have cloned the animation controller, the original animation controller can be released.

We clone an animation controller using the ID3DXAnimationController::CloneAnimationController function shown next. It is nearly identical to the D3DXCreateAnimationController function, so the parameter list should require no explanation.

```
HRESULT CloneAnimationController
(
    UINT MaxNumAnimationOutputs,
    UINT MaxNumAnimationSets,
    UINT MaxNumTracks,
    UINT MaxNumEvents,
    LPD3DXANIMATIONCONTROLLER *ppAnimController
);
```

> **Note:** We will cover events (parameter four) later in this chapter when we cover the animation sequencer.

10.8.5 Hierarchy Animation with ID3DXAnimationController

Once we have the animation controller that suits our needs, we are ready to use it to animate our hierarchy. While we have looked at quite a few functions for setup and configuration for both our animation controller and animation mixer, once all of that is done, actually playing the animations could not be easier. In fact, there is really just a single function (AdvanceTime) that we need to use.

AdvanceTime takes as its first parameter the time (in seconds) that has elapsed since the last call to AdvanceTime (which will be zero the first time we call it) and then updates the internal timer of the controller and the timer of each animation track (scaled by track speed). It then maps the timer of each track into the periodic position of its assigned animation set and uses this periodic position to call the animation set's GetSRT method for each animation it contains to generate the new SRT data for the frame to which it is attached. If multiple tracks (animation sets) are being used, then it is possible that there may be an animation in each animation set that animates the same frame in the hierarchy. In that situation we know that there will be multiple sets of SRT data for that single hierarchy frame. The animation mixer will blend those SRT sets together to calculate the final transformation matrix for that frame based on the properties we discussed earlier. The AdvanceTime function is shown next. Please note that this function takes a second optional parameter which allows us to pass in a pointer to a callback function. We will discuss the callback feature of the animation system at the end of this chapter, so for now just assume that we are passing in NULL as this parameter.

```
HRESULT AdvanceTime( DOUBLE Time,
                     LPD3DXANIMATIONCALLBACKHANDLER pCallbackHandler);
```

Simply put, once the animation controller has been configured, playing animation involves nothing more than calling this function in your game update loop, passing in the time that has elapsed since the previous call. The next code example uses our timer class to get the number of seconds that have passed since the last call and passes it into the AdvanceTime method of the animation controller. After this call, all the parent-relative matrices in the hierarchy will have been updated.

```
void MyClass::UpdateAnimations ( )
{
      double ElapsedTime =  m_Timer.GetTimeElapsed() ;
      m_pAnimController->AdvanceTime( ElapsedTime );
}
```

This single function call will calculate all of the current animation data and update the hierarchy frame matrices so that our meshes can be scaled, rotated, and translated according to the animation being played. Once the above function has been called, the hierarchy is ready to be traversed (where we will calculate the world matrices) and rendered as discussed earlier. Our approach to rendering the hierarchy is not altered at all by the fact that we are animating it. The animation controller simply changed the values stored in the relative matrices of our frame hierarchy so that the next time it is updated and rendered the meshes appear in their new positions.

If your animations are set to loop, you do not have to worry about keeping a running total of the sum of elapsed time or wrapping back round to zero after exceeding the length of the longest running animation set. This is because once the track timers are updated, they are mapped into the periodic position of the assigned animation set. If an animation set is configured to loop or ping-pong, then regardless of how high the value of the track's timer climbs, it will always be mapped into a periodic position within the bounding keyframes of the animation set.

If your animation sets are not configured for continuous play, then some care will need to be taken when calling AdvanceTime. Once a track timer exceeds the period of its animation set, the animation set will cease to play (technically, it has ended at this point). Of course, we have seen that you have the ability to set the position of a track back to zero to cause this animation set to play again, even if the global time exceeds the length of the animation set. It is helpful that the global time is decoupled from the track timers in this way.

As an example, imagine that we have an animation set with a period of 15 seconds which is not set to loop. It also has a track speed of 1.0, so initially there is a 1:1 mapping between global time, track time and the periodic position of the animation set. We might decide to perform out own looping logic so that we can control when the animation set starts playing. Imagine that although the animation set has a period of 15 seconds, we only want it to start playing again at a global time of 20 seconds, thus causing 5 seconds of inactivity between the time the first animation loop ends and the second one begins. The following code would achieve this (note that it assumes that the animation set is assigned to track 0 on the mixer):

```
void ProcessAnimations ( double ElapsedTime )
{
      static double WrapTime  = 20.0;

      double GlobalTime = pController->GetTime();
```

```
      if ( GlobalTime >= WrapTime )
      {
            pController->SetTrackPosition( 0 , GlobalTime-WrapTime );
            WrapTime+=20.0f
      }

      pController->AdvanceTime ( ElapsedTime );
};
```

Notice the call to the controller's GetTime method. This is a new method which we have not yet discussed. GetTime returns the global time of the controller. Remember, the global time is really the sum of all elapsed times that have been passed into our AdvanceTime calls since the animation began.

`DOUBLE GetTime(VOID);`

The previous code initially sets the wrap around time at 20 seconds global time. When this time is reached, the track timer is wrapped around to zero and a new wrap around time of 40 seconds is set. At 40 seconds the track timer will be wrapped around to zero and a new global wrap around time of 60 seconds would be set, and so on. While the period of the animation set is only 15 seconds and is not configured to loop, the above code would reset the track timer to 0 at every 20 second global time boundary. This clearly shows the distinction between the global timer which is ever increasing with each AdvanceTime call, and the track timers which can be reset, positioned or stopped independent of the global timer.

For no other reason than to better understand this concept, let us take a quick look at how we might write a simplified AdvanceTime function of our own that works with multiple animation sets. This will let us see how the timers are incremented. We will not worry about animation mixing in this example and will assume that each animation set influences its own section of then hierarchy. This will give us a rough idea of what is happening behind the scenes in the D3DXAnimationController::AdvanceTime function. Obviously there is a lot missing here, mainly the mixing of multiple frame SRT sets and the callback and sequencing systems which we will discuss shortly. However, this example does allow us to get the basic idea of how the update works and how the controller interacts with each track and its assigned animation sets. Fortunately, we will never have to write a function like this since the D3DX animation controller does it all for us.

```
void MyAnimController::AdvanceTime( double Elapsed )
{
    LPD3DXKEYFRAMEDANIMATIONSET pAnimSet = NULL;
    LPD3DXFRAME pFrame = NULL;

    D3DXVECTOR3 Scale, Translate;
    D3DXQUATERNION Rotate;
    D3DXMATRIX mtxFrame;
    ULONG i;
    char AnimName [1024];

    m_GlobalTime+=ElapsedTime;

   // Loop through each track
   for ( uint TrackIndex = 0; TrackIndex < GetNumTracks() ; TrackIndex++ )
```

```
    {
       // Increment track timer scaled by track speed
       m_Track[TrackIndex].TimerPositon+=ElapsedTime*m_Track[TrackIndex].Speed;

       // Get Anim set assigned to this track
       pAnimSet = GetAnimationSet( TrackIndex , &pAnim );

       // Map track time to animation set time
       float PeriodicPosition = pAnimSet->GetPeriodicPosition(
                                              m_Track[TrackIndex].TimerPosition );
       // Get the number of animations in this animation set
       ULONG NumAnimations = pAnimSet->GetNumAnimations();

       // Loop through each animation in this animation set and get the new SRT data
       for ( Animation = 0; Animation < NumAnimations; ++Animation )
       {

          // Get name of current animation
          // should match the name of the frame it animates
          pAnimSet->GetAnimationNameByIndex( Animation , &AnimName );

          // Generate the Scale, Rotation and Translation data for this animation
          // for the current periodic position
          // We looked at GetSRT earlier.
          pAnimSet->GetSRT( PeriodicPosition, Animation ,
                            &Scale, &Rotate, &Translate );

          // Generate a matrix from the SRT data.
          // This global D3DX function generates a 4x4
          // transformation matrix from a scale vector, a quaternion
          // and a translation vector. Check out the docs for details.
          D3DXMatrixTransformation( &mtxFrame, NULL, NULL, &Scale,
                                    NULL, &Rotate, &Translate );

          // Find the frame that this interpolator animates in the hierarchy
          pFrame = D3DXFrameFind(m_pFrameRoot, AnimName );

          // Update the frames matrix with out new matrix
          pFrame->TransformationMatrix = mtxFrame;

     } // Next Animation

     // Release the anim set we no longer need it
     pAnimSet->Release();

   } // Next Track
}
```

The first thing this function does is update the global time of the controller. This global timer may seem useless at the moment, but when we examine callbacks, you will see how the global timer becomes a much more useful feature. Next, we loop through each track and increment its timer. In this case we scale the elapsed time about to be added by the speed property of the track. We then convert the new track time into an animation set periodic position which we can feed into the GetSRT function. We then loop through each animation stored in the animation set and, for each one, call ID3DXAnimationSet::GetSRT to generate the new scale, rotation and translation information. We then

use the returned information to construct a matrix. Once the matrix is constructed, we search the hierarchy for a frame with the same name as the animation and replace its frame matrix with the newly generated one. The D3DX animation controller would not search the hierarchy in this way as it would be far too slow, which is why it stores the matrices in a separate array that can be accessed quickly. Indeed, this is why we register animation outputs in the way discussed earlier -- so the controller can know exactly which matrix is paired with which animation and quick matrix updates can be performed without the need for frame hierarchy traversal. However, as this function has been written with clarity of process in mind, we will forgive ourselves for bypassing this optimization.

One thing you may have noticed is that we are not multiplying the current frame matrix by the newly generated animation matrix. This is because all of the keyframe animation data stored inside an animation is ***absolute***. This is true of both the data in the .X file itself as well as in the data output by the SRT functions (by definition, because it is stored this way in the file). To be clear, what is meant by 'absolute' is that the keyframe data is not relative to what came before it (i.e. it is not incremental). Each keyframe is a complete snapshot that describes the transformation state of the object at that particular moment on the timeline. The transformations are still defined relative to the parent frame of course (it's still a parent-relative matrix), because we want the spatial hierarchy relationships to remain intact during animation. This means that we are able to overwrite the frame matrix without fear of having to preserve or take into account previous transformations. Each time we call AdvanceTime, the frame matrix will be repopulated with the next set of absolute SRT values. In other words, we get a new frame matrix for each animation update. To make the system work incrementally would be more challenging and would require that we track more data between update calls. Since this is typically unnecessary and does not add much value to the system, absolute values are preferred.

> **Note:** Although we are searching through the frames here (using D3DXFrameFind) to find the correct matrix in the hierarchy to modify, the animation controller does not need to do this. Recall that it has access to a pointer for each matrix in the hierarchy along with the name of the frame it is associated with. For more information you can refer back to our discussion of the ID3DXAnimationController::RegisterAnimationOutput function discussed earlier.

ResetTime

The final function we will cover in this section is called ResetTime. It allows the application to wrap the global time of the animation controller back to zero, much like we did in the earlier code with the track timers. This is especially handy if we have many animation sets that do not loop and would cease playing once the sum of elapsed time passed into the AdvanceTime method exceeded its period. This function causes the entire animation system to restart, taking smooth wrapping of track timers into account. This is especially useful when considering sequencing which we will discuss in the next section.

For now, just know that sequencer events (called key events) are registered on the global controller timeline (not the per-track timelines) and as such, once an event has been triggered when the global time has reached its timestamp, it will never be triggered again unless the global time is restarted. So we can imagine that a complex group of animation sets might wish to be looped at the global level so that the key events set on the global timeline are also re-triggered each time. If we simply configured each

animation set to loop, the animations themselves would loop but each key event on the global timeline would only be triggered once.

It is also worthy of note that one might also choose to reset the global time of the controller even if the sequencer is not being used, simply to stop the global time value reaching an extremely high value. Obviously this would only be a concern if an animation was left to play for a considerable period of time. The method is shown below.

```
HRESULT ResetTime(VOID);
```

The ResetTime method is actually 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 as 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.

During the call to ResetTime, the animation set's GetPeriodicPosition function is called, passing in the current track time. If the D3DXTRACK_DESC::Position was 137 prior to ResetTime(), then this is the position that gets passed to the GetPeriodicPosition call. Normally, GetPeriodicPosition returns the current time, after it has been mapped into the set's local timeline. In the case of looping for example, we can assume that this is a simple fmod() of the current track time with the period of the animation set as discussed earlier. During ResetTime, the information returned by GetPeriodicPosition is taken as a 'snapshot', and all internal time data is updated using this snapshot. The information returned by GetPeriodicPosition is used to update the track position. You can prove this by examining the track timer (its position) both before and after the ResetTime call. Meanwhile the controller's global time is reset to 0.0.

To summarize, let us look at an example of what is happening here. Imagine we had an animation set with a period of 10 seconds and let us make the call to ResetTime after 35 seconds of application time have elapsed. Prior to the ResetTime call, both our track containing the animation set, and our global time contain the value 35.0 (assuming we maintained a constant track speed of 1.0). At this point we make our call to ResetTime. During the call, the animation set's GetPeriodicPosition function is called and it returns what is essentially fmod(TrackPosition, pAnimSet->Period) -- therefore a value of 5.0. Once the call has completed, we find that the controller's global time has been reset to 0.0, and the track position has been merely wrapped back into a sensible range, and now contains a value of 5.0.

> **NOTE:** No other animation set function is called during the Reset procedure. In addition, there are implications with respect to the callback mechanism. We will discuss this later when examining the D3DX animation callback mechanism.

So we have now looked at how to advance our animations during runtime using a simple function call. We have also seen how we can retrieve the global time of the controller if we wish to allow the application the freedom to manipulate a certain track at key global times. However, while we could allow our application to control the track properties at certain global times using the method shown earlier, this is precisely what the animation sequencer is for, and is the subject of our next section.

10.9 The Animation Sequencer

Earlier in the chapter we mentioned that the animation controller contains a very useful event scheduler, referred to as the **sequencer**, which allows us to apply and adjust track properties at user defined global times. In this section we will discuss the animation sequencer; another key component of the D3DX animation controller. What we will see is that the global timer of the controller is more than just a counter for elapsed time passed into AdvanceTime calls; it is actually the heartbeat of a powerful tool.

Recall that the elapsed time that we pass to AdvanceTime increments both global and local timelines. The global time is maintained at the controller level and is the value that is returned in response to a GetTime function call. It is ultimately the amount of time that has passed in seconds since calls to AdvanceTime began (assuming that you have not called ID3DXAnimationController::ResetTime to reset the global time back to zero). Unlike the animation set's GetSRT function which uses track times, the sequencer works using the controller's global time value of the controller.

For example, let us imagine that we would like to schedule an event that disables an animation track at the global time of 12 seconds. Further, let us assume that the animation set we are working with is only 10 seconds in duration, but is configured to loop. In this example, the animation set would stop playing exactly 2 seconds into its second iteration. This event would trigger only occur once -- when a time of 12 seconds or greater is first reached on the global timeline. This same event will *not* be triggered again with further calls to AdvanceTime (unless ResetTime is called).

If we want an event to be periodic (i.e., triggered once every N seconds), then the application is responsible for providing the logic that wraps the global time before it calls AdvanceTime again. For example, using our 12 second (global time) trigger described above, if we were calling the following function repeatedly (passing in the time in seconds that has elapsed since the last frame), the event would initially be triggered 12 seconds into the animation loop. Notice however, that every 20 seconds we wrap the time back to zero. Thus, after the first time the event has been triggered, it is then re-triggered every 20 seconds thereafter (i.e. every time CurrentGlobalTime reaches 12 seconds again).

```
void AnimateHierarchy ( double TimeElapsed )
{
    CurrentGlobalTime = pAnimController->GetTime();

    if ( CurrentGlobalTime >= 20.0 ) pAnimController->ResetTime();

    pAnimController->AdvanceTime( TimeElapsed );
}
```

Just try to remember that events are scheduled in global time while animation sets use track time to loop independently. If your track speed is 1.0 and you have not altered the position of the track, then a 1:1 mapping will be maintained between global time and track time.

Let us look at another example of why this is useful. We will assume that we have a hierarchy that contains the mesh of a space station and the mesh of a space ship, and that we have defined two looping animation sets. The first animation set slowly rotates the space station frame along its local 'up' axis.

The second animation set is a 20 second animation set that animates the frame of the space ship so that it starts off docked inside the space station and then flies out of the space station and off into deep space (perhaps through a jump gate or wormhole). This sequence will make a space simulation more realistic as we come in to dock our ship, since we will see other space ships making use of the station as well. As you might imagine, simply looping the space ship animation set presents a bit of a problem. Seeing a new space ship emerge from the space station precisely every 20 seconds makes it quite obvious to the player that this is just a looping animation. We will generally prefer a less uniform and more random series of events as would be the case in the real world (using the term loosely of course given the circumstances). Making use of our sequencer provides an opportunity to improve our situation. We could very easily make the pattern a lot less obvious simply by choosing a more random distribution of global times to start and stop the animation (i.e. enable/disable the track). Imagine for example, that we set events for the second track on the mixer (the track our space ship animation is assigned to) to occur at the following global times (in seconds). Keep in mind that the space ship animation lasts for 20 seconds before looping.

Event Number	Global Time (GT)	Event Type
1	20	Track Disable
2	30	Track Enable
3	70	Track Disable
4	100	Track Enable
5	160	Track Disable
6	180	Track Enable

Our accompanying animation update function might look something like the following:

```
void AnimateSpaceSceneHierarchy(FLOAT TimeElapsed)
{
       if ( pAnimController->GetTime() > 200.0 )  pAnimController->Reset();
       pAnimController->AdvanceTime( TimeElapsed );
}
```

We see immediately that our sequencer loop is now on a much larger scale of 200 seconds with some event distribution in between. This should break up the pattern nicely over time and provide some additional realism. Here is how everything would play out…

The space ship track would start off enabled and the space ship animation would play out to completion once (it lasts for 20 seconds). Our first event occurs immediately (GT = 20 secs) which turns off the track, creating a 10 second gap where no animation is being played for the space ship. 10 seconds later (GT = 30 secs) our second event occurs which re-enables the space ship track. Note that during the 10 second time between the first two events, the animation controller will ignore the disabled track and will not increment the track's timer when AdvanceTime is called. Therefore, our space ship mesh would be frozen somewhere off in the distance (hopefully far enough away so as not to be visible!). Notice as well that we are very careful to disable the track on 20 second boundaries so that we do not disable the animation while the space ship is still in the middle of its sequence.

The track then plays for 40 more seconds causing the animation to play two more times (looping) before our next event occurs (GT = 70 secs). This third event disables the track again. Because the animation is

always being frozen on a 20 second boundary, we can be sure that whenever we enable it again it will start off at a track position close to zero where the ship is back inside the space station. Our fourth event (GT = 100 secs) re-enables the track and we let the track play for another 60 seconds. This means that the animation will loop three times, one immediately after another. We then disable the track again at event five (GT = 160 secs) and do not enable it again until our sixth event (GT = 180 secs) leaving a 20 second gap where no ships leave the space station. Finally our sixth and last event (GT = 180 secs) re-enables the track. Note that the looping would remain enabled and we would see ships take off every 20 seconds were it not for the logic in our AnimateSpaceSceneHierarchy function. Rather than continue to increment the global time, we roll it back to zero once we have had 200 seconds elapsed in total. Thus, the whole cycle would start again. Our final event which re-enables the space ship animation would actually play twice before being disabled again at the rolled around global time of 20 seconds.

Admittedly, this very simple example only scratches the surface of some of the creative ideas you can come up with for using the animation sequencer. But it is clear enough that just by registering a few track enable/disable events with the sequencer we have turned a 20 second looping animation into a 200 second animation sequence that is far less uniform and perceptible to the player. The gap between ships leaving the station would no longer seem fixed and it is unlikely that the player would pick up on this pattern. You could probably imagine experimenting with additional values like track speed as well so that the ships seem to fly faster or slower. Perhaps you might occasionally turn on blending with a second track that includes a different translation keyframe value so that ships fly off in different directions. Maybe you even would turn on an afterburner track at random intervals, etc.

With this high level overview in place, let us now examine how we can set key events on the global timeline using the controller's sequencing methods. We will also need to discuss the types of key events we can register.

10.9.1 Registering Sequencer Events

There are five different Event Key types that we can register with the animation sequencer. This is done via one of five corresponding methods exposed by the ID3DXAnimationController interface. So to set an event, we just call the function which matches the event type and we are done. It is very important that we do not try to register more events with the sequencer than the maximum number of events that the animation controller can process. Recall that we set this maximum number of events when we created/cloned the animation controller (the MaxNumEvents parameter).

> **Note:** D3DXLoadMeshHierarchyFromX creates an animation controller that can store a maximum of 30 events. Trying to register additional events beyond those 30 will fail. To increase the event pool size, we simply need to clone a new controller and specify the desired number of slots.

We can query the number of events the animation controller can have registered with it using the following method of the ID3DXAnimationController interface. This next function returns the maximum number of events that can be registered with the controller.

```
UINT GetMaxNumEvents(VOID);
```

Let us quickly discuss these five functions along with their parameters and the types of events they allow us to set. Notice how each of the event registration functions returns a D3DXEVENTHANDLE. We will discuss what these handles can be used for at the end of this section.

1. The KeyTrackEnable Function

This function allows us to schedule an event that will enable or disable a track on the animation mixer (a 'KeyTrackEnable' event). When a track on the mixer is disabled, any animation set assigned to that track is essentially frozen. It will not receive any updates from the AdvanceTime function and the track timer will not be incremented until it becomes re-enabled. Note that when we re-enable the track, it will not restart from the beginning -- it will resume from the time it was frozen. This allows us to freeze an animation at any point and then resume it whenever we wish.

```
D3DXEVENTHANDLE KeyTrackEnable(UINT Track, BOOL NewEnable,
                               double StartTime);
```

UINT Track
This parameter is the zero-based index of the mixer track we are registering the event for. For example, to disable the fifth track of the animation mixer (and its assigned animation set) at a specific global time, we would call this function with a value of 4 for this parameter.

BOOL NewEnable
This second parameter should be set to true or false depending on whether we wish to enable or disable the specified track at the specified time, respectively.

DOUBLE StartTime
The final parameter is the global time (in seconds) when we would like this event to occur.

Example:

```
pAnimController->KeyTrackEnable( 0 , FALSE , 35.0 );
pAnimController->KeyTrackEnable( 2 , TRUE , 100.0 );
```

The prior example registers two KeyTrackEnable events with the animation sequencer. In the first call to the function we specify that we would like the first track on the animation mixer (track 0) to be disabled 35 seconds into the animation (when the sum of elapsed time passed into the AdvanceTime function passes 35 seconds). In the second call to the function we register an event with the sequencer that specifies that at 100 seconds global time, we would like the third track of the animation mixer (track 2) enabled and to start playing its assigned animation set.

2. The KeyTrackPosition Function

The next type of event that we can register with the animation sequencer is referred to as a KeyTrackPosition event. This event allows us to set the timer of a track to a specific value (position)

when a specific global time is reached. In other words, we are saying that at a specific global time, we would like to instantly transition the track timer to a specific value regardless of the current global time. As it is the position of a track (the track timer) that is used to map to a periodic position, this function ultimately allows us to jump to a desired location in the animation set at a specified global time.

This can be a very useful event if you are using a single animation set that stores all of your animation sequences (ex. run, walk, jump, etc). Imagine that we had an animation set that contained only two animation sequences: idle and walk. You could quickly pre-script some AI-like events (or load them from a file) that basically has a character which idles for a few seconds and then starts walking, stops and idles again, walks a little more, etc. While we will do this differently in our demo engine (using separate animation sets and then enabling/disabling their tracks), if you are using single animation sets, this might be an approach to consider. Note that if you wanted your AI to make these decisions on the fly, rather than use a pre-programmed set of events, you would probably use the ID3DXAnimationController::SetTrackPosition to cause the timeline advancement to happen immediately. However, there are certainly cases where the event management concept can be put to good use. Your application could, for example, decide to register a track position change event to occur in 10 seconds time using the sequencing functions.

```
D3DXEVENTHANDLE KeyTrackPosition( UINT  Track, double NewPosition,
                                  double StartTime );
```

UINT Track
This is the index of the mixer track for which we would like this event to apply. It is the timer of this track which the event will manipulate.

double NewPosition
This parameter contains the new position that we would like this track's timer set to. For example, if this value was set to 200, then when this event is triggered, the track time would instantly be set to 200 and any future calls to AdvanceTime would add elapsed time relative to this. This advancement would happen at the global time specified in the StartTime parameter. Remember also that this is not the same as directly specifying a position in the animation set (although it can be). The track position is mapped to a periodic position so that looping is factored in, and it is the periodic position that is used to fetch the SRT data for each animation. Therefore, manipulating the timer of the track allows us to *indirectly* manipulate the periodic position generated for the animation set.

double StartTime
This is the global time (in seconds) when we would like this event to occur. When the global timer of the controller reaches or passes this value, the event will be triggered.

As another simple example, we might imagine that we have a zombie in our game that, five seconds after being hit by our bullet, disintegrates into a pile of dust and bones. Again, we are assuming that this disintegration sequence is part of a single animation set that might include other animation sequences like the zombie walking or attacking, etc. In the following example, the zombie animation set is assigned to track 0 on the animation mixer. When the zombie is hit by a bullet we register an event that advances the animation to the death sequence (position 200) in 5 seconds time (5 seconds being an arbitrary delay that we have chosen to delay the disintegration sequence).

```
if ( ZombieIsHitByBullet )
{
      double CurrentTime = pAnimController->GetTime ();
      pAnimController->KeyTrackPosition ( 0 , 200 , CurrentTime + 5.0 );
}
```

3. The KeyTrackSpeed Function

We mentioned earlier how the ID3DXAnimtionController::SetTrackSpeed function sets a scalar value (per track) to scale the amount that the track's local time is incremented using the passed global time. Setting this to 1.0 means that the animation set time is incremented normally. So we can see that each track has a clock and a speed value. We can set this speed value directly using the SetTrackSpeed function discussed earlier in the chapter or we can choose to let the sequencer do it at a certain global time using the KeyTrackAnimSpeed function.

```
D3DXEVENTHANDLE KeyTrackSpeed ( UINT Track, float NewSpeed,
                                double StartTime, double Duration,
                                DWORD Method );
```

DWORD Track
This is the index of the track we would like the event to be applied to.

float NewSpeed
This is the new speed value which we will set for this track. A value of 1.0 means the animation will play at its default speed (the speed intended by the animation creator). The speed value of each track is used to scale the amount by which the track's internal clock (i.e., the track position) is incremented. Setting this value to 3.0 will cause the animation controller to scale the elapsed time passed into the AdvanceTime method by 3, before adding it to the track timer.

double StartTime
As with all key event types that we can register with the sequencer, we always have to specify a global time at which this event will take place. If we set this value to 200, it means that this event will be triggered when the controller global timer reaches 200 seconds.

double Duration
Unlike the previous functions we discussed, this function allows us to specify an end time for our requested change so that we can transition gracefully between events. The `Duration` parameter specifies how long (in seconds) we want the transition between the old speed and the new speed to take. For example, assume that the current speed of the track is 10.0 and we set an event at 100 seconds global time to set the speed to 5.0. We also set the duration of the event to 10.0. Then starting at 100 seconds global time, the speed of the animation set assigned to this track would slowly decrease over the next 10 seconds until it finally reached the requested speed of 5.0. In other words, it would take 10 seconds to reach the requested speed of 5.0 -- we gradually ease in to it rather than abruptly change. The calculation used to scale between the current speed and the requested speed of the track is determined by the Method parameter discussed next.

DWORD Method
We use this parameter to pass a member of the `D3DXTRANSITIONTYPE` enumerator. It describes the method we would like to use to run the transition calculation between current and target values. There are two methods to choose from: `D3DXTRANSITION_LINEAR` provides a simple linear interpolation between the old speed and the new speed and `D3DXTRANSITION_EASEINEASEOUT` uses a spline-based curve to handle the transition. In other words, it is linear falloff versus exponential falloff (to refer back to some familiar material we discussed in Chapter Five).

Example

```
pAnimController->KeyTrackSpeed( 2 , 2.0 , 200.0 , 10.0 , D3DXTRANSLATION_LINEAR );
```

This example sets an event for the third mixer track (track 2) to be triggered at 200 seconds global time. It will begin to set the speed value for the track to 2.0, from whatever its current speed is, over a period of 10 seconds. Linear interpolation will be used to affect the changes in between.

4. The KeyTrackWeight Function

Every track on the animation mixer can be assigned a blend weight using the ID3DXAnimationController::SetTrackWeight function discussed earlier. This weight is used to scale the influence of the animation set on the respective frames in the hierarchy. We can also schedule a KeyTrackWeight event with the sequencer so that the weight of a track can be changed at scheduled global times.

Like the previous function discussed, we are able to apply our change over a specified duration. As one usage example, we might have an animation that violently shakes certain meshes in the hierarchy in response to a laser blast. We could schedule the weight of this 'shake' animation to decrease to zero over a period of N seconds, such that the effect of the blast on the hierarchy seems to dissipate with time (i.e. less and less noticeable shaking over time).

```
D3DXEVENTHANDLE KeyTrackWeight ( DWORD Track, float NewWeight,
                                 double StartTime, double Duration,
                                 DWORD Method );
```

DWORD Track
This is the track that we wish this event to be scheduled for.

float NewWeight
This is the new weight that we would like to set for this track. The default value is 1.0, so the animation affects the hierarchy at full strength. Setting this value to 0.5 for example would mean that the animation would affect the hierarchy at only half strength.

double StartTime
This is the global time when the event should occur.

double Duration
This is the span of time over which the transition from the previous weight to the new weight should take place. If we specified a new weight of 0.0 and the current weight was 1.0, and we used a Duration of 3 seconds, then once the specified global time was reached, the weight of the track would be slowly degraded from 1.0 to 0.0 over a period of 3 seconds.

DWORD Method
This parameter indicates the interpolation method that generates the intermediate values during the transition between current and target weight values. The two choices are linear interpolation (`D3DXTRANSITION_LINEAR`) or interpolation along a curve (`D3DXTRANSITION_EASEINEASEOUT`).

```
pAnimController->KeyTrackWeight(1, 0.0, 100.0, 3.0, D3DXTRANSITION_EASEINEASEOUT);
```

This example schedules a weight event to take place on track 1 at 100 seconds global time. The weight will be exponentially scaled down from its current weight to a value of 0.0 over a period of 3 seconds. Once done, the animation assigned to this track will not affect the hierarchy in any way.

5. The KeyPriorityBlend Function

The final event type we can register with the sequencer is the priority blending event. This is the sequencer replacement for the `SetPriorityBlend` function covered earlier. This function allows us to smoothly transition between weighting the blending between high and low priority track groups. The function is shown below with a list of its parameters. Notice that unlike the other sequencing functions we have discussed, there is no Track parameter. You should recall that this is because the priority blend is a property of the animation mixer itself, and not a specific track.

```
D3DXEVENTHANDLE KeyPriorityBlend( float BlendWeight,
                                  double StartTime, double Duration,
                                  DWORD Method );
```

float BlendWeight
This parameter is the new priority blend value for the animation mixer. It should be a value between 0.0 and 1.0. This value can be thought of as setting the strength of low priority tracks in the final blending operation. A value of 1.0 would mean that the high priority tracks do not influence the hierarchy at all.

double StartTime
The global time in seconds at which the event should be triggered.

double Duration
The length of time it should take to transition between the current and target priority blend weights.

DWORD Method
This parameter indicates the interpolative method that generates the intermediate values during the transition between current and target priority blend weight values. The two choices are linear interpolation (`D3DXTRANSITION_LINEAR`) or interpolation along a curve (`D3DXTRANSITION_EASEINEASEOUT`).

10.9.2 Event Handles and Working with Registered Events

We have now seen how easy it is to register an event with the sequencer; a simple function call and the work is done. However, there may be times when you wish to reset the global time of the controller back to zero, but do not wish a certain event to occur a second time through. In order to address this situation, we need a function that allows us to remove an event from the global timeline as well as register one. The ID3DXAnimationController provides us such a method. It also provides methods for querying upcoming events, query current events being executed, and validating events as still active or not.

In order to ask the animation controller for details regarding an event, we need some way to inform it which event we are enquiring about. You will likely have noticed that each of the functions that register a key event returns a D3DXEVENTHANDLE. This handle can be stored and used to enquire about that event at a later time.

HRESULT UnkeyEvent(D3DXEVENTHANDLE hEvent);

This function allows the application to remove a registered event from the sequencer. You simply pass in the handle of the event. Below we see an example of a TrackEnable event that has been registered to occur at 100 seconds global time. We then use the returned handle to later remove it from the timeline. Ultimately, this allows us to discard events which we no longer need, which can be useful if your animation controller has been configured to store only a small number of events.

```
D3DXEVENTHANDLE Handle;
Handle = pAnimController->KeyTrackEnable( 2 , TRUE , 100.0 );
..
..
..
pAnimController->UnkeyEvent( Handle );
```

As you can see, the handle returned from the KeyTrackEnable function is used to identify the event to the controller in the UnkeyEvent method.

HRESULT UnkeyAllTrackEvents(UINT Track);

This function does not require an event handle to be passed in its parameter list because it is used to remove *all* events scheduled for a given track. This will not affect any events that have been registered for other tracks. Of course, it cannot be used to unregister priority blend events, since priority blend events are global to the controller and are not assigned to any track. There is a separate function (discussed next) for removing all events of this type.

In the following example we register three events with the controller. Two of these events are registered for track 2 on the mixer at 100 and 300 seconds respectively, and an event is also registered for track 3 to occur at a time of 200 seconds.

```
pAnimController->KeyTrackEnable( 2 , TRUE , 100.0 );
pAnimController->KeyTrackEnable( 3 , FALSE ,200.0 );
pAnimController->KeyTrackEnable( 2 , TRUE , 300.0 );
…
…
pAnimController->UnKeyAllTrackEvents( 2 );
```

Notice that after the three events have been registered, later in the code we unregister all events assigned to track 2. In the above example, this would remove only two of the three events. The event that was registered for track 3 at a global time of 200 seconds would still remain on the global timeline.

HRESULT UnkeyAllPriorityBlends(VOID);

When priority blend events are scheduled using the KeyPriorityBlend function, these events alter a global setting of the controller, and as such are not specific to any track. Therefore, if we wish to remove all priority blend events that have been registered with the animation sequencer, we must use the UnkeyAllPriorityBlends function. This function will remove *all* events that have been previously registered with the KeyPriorityBlend function.

HRESULT ValidateEvent(D3DXEVENTHANDLE hEvent);

The ValidateEvent function can be used to test if an event is still valid. Valid events are events which either have not been executed yet (because the global time is less than the event timestamp) or events that are currently being executed.

In this next example we show an event that has been registered for track 2 at 100 seconds global time. Elsewhere in the code, the event is validated. If the event is found to be invalid then it means it has already been triggered and has completed. In this case, we then remove the invalid event from the timeline.

```
D3DXEVENTHANDLE Handle;
Handle = pAnimController->KeyTrackEnable( 2 , TRUE , 100.0 );
..
..
..
if ( FAILED ( pAnimController->ValidateEvent( Handle )) )
     UnKeyEvent ( Handle );
```

This method provides a way for our application to know whether a certain event has been triggered yet. However, you should be careful if you intend to reset the global time and wish the events to be triggered again each time through the global timeline. Once the event is gone, it is gone for good. If the application was then to call the ResetTime method resetting the global time of the controller to zero, the same event would not be triggered when the global time reaches 100 seconds (in the above example) a second time because it has been removed from the timeline

```
D3DXEVENTHANDLE GetCurrentTrackEvent( UINT Track,
                                      D3DXEVENT_TYPE EventType );
```

This function allows you to query the current event that is running on a particular track. This allows the application to fetch the event handle that is currently being executed. This event handle can then be passed into the GetEventDesc method (discussed below) to fetch the event details.

For the first parameter, the function is passed the index of the track for which we wish to fetch the handle of the event currently being executed. The second parameter allows us to specify a filter so that we can fetch the event only if it is of a certain type. If no event of the specified type is currently running on the specified track, the function will return NULL.
For the second parameter, we should pass in a member of the D3DXEVENT_TYPE enumeration:

```
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 in D3DXEVENT_PRIORITYBLEND to this function will always return NULL because a priority blend event is not a track event. The current code shows how we might query for the currently running track speed event (if any) on track 2 and output the speed value. This could be useful for debugging or perhaps for slowing down in-game MIDI music based on the speed of a track.

```
D3DXEVENTHANDLE Handle;
Handle = pController->GetCurrentTrackEvent( 2 , D3DXEVENT_TRACKSPEED );

if ( Handle )
{
     D3DXEVENT_DESC desc;
     PController->GetEventDesc( Handle , &desc );
     Double Speed = desc.Speed;

     if ( Speed < 0.5 ) PlaySlowMusic();
     else
                        PlayFastMusic();
}
```

In the above pseudo-code, the handle of any currently playing speed change event is requested. If a speed event is currently being triggered on track 2, we are returned the handle (or NULL if no speed event is currently being carried out).

If an event is being performed, then we use the GetEventDesc method of the controller (discussed shortly) to fetch the details of the event. They are returned in the passed D3DXEVENT_DESC structure. In this example, the speed of the event is queried and used to play different music depending on the speed at which the animation is playing.

```
D3DXEVENTHANDLE GetCurrentPriorityBlend(VOID);
```

This method is the sister function of the function previously described. As discussed previously, priority blend events are different from all other key events in that they are not assigned to a specific track. As such, a different function is exposed to allow the application to retrieve the event handle of any priority blend event that is currently being executed.

The function takes no parameters and returns the event handle of the current priority blend event being executed by the controller (or NULL if no priority blend event is currently in progress).
Once you have the event handle, you can then call the GetEventDesc method (just as we did in the prior code example) to fetch the details of the priority blend event. This will tell us things like its time stamp, its start time, weight, and duration.

```
HRESULT GetEventDesc( D3DXEVENTHANDLE hEvent,  LPD3DXEVENT_DESC pDesc );
```

This function is useful for extracting the information for any event for which the application has an event handle. By simply passing in the event handle as the first parameter and a pointer to a D3DXEVENT_DESC structure, on function return, the details of the event will be stored in the passed structure.

We used this function in the description of the GetCurrentTrackEvent example above, but it can also be used to retrieve the information of a priority blend event (if the handle to a priority blend event is passed in as the first parameter).

The D3DXEVENT_DESC will contain all the information about the event:

```
typedef struct D3DXEVENT_DESC {

    D3DXEVENT_TYPE      Type;
    UINT                Track;
    DOUBLE              StartTime;
    DOUBLE              Duration;
    D3DXTRANSITION_TYPE Transition;

    union
    {
        FLOAT  Weight;
        FLOAT  Speed;
        DOUBLE Position;
        BOOL   Enable;
    }

} D3DXEVENT_DESC, *LPD3DXEVENT
```

As we might expect, the first five members describe the details of the event itself (e.g. what type of event it is -- D3DXEVENT_TRACKSPEED or D3DXEVENT_PRIORITYBLEND). We are also returned the track number that the event applies to (not applicable to priority blend events) and the global time for the event trigger and the duration (in seconds) that the event will take to fully execute. We can also examine the transition type that is being used for this event to reach its desired final state.

We discussed the various events that can be scheduled earlier, and saw that duration and transition information is not applicable to KeyTrackEnable events or KeyTrackPosition events.

Finally, the last member of this structure is a union. The member that is active in this union depends on the type of event that it is. For example, if the Type parameter is set to D3DXEVENT_TRACKWEIGHT or D3DXEVENT_PRIORITYBLEND then the Weight member of this union will contain the Weight that this event will set the track to use or set the priority blend to. If the event type is D3DXEVENT_TRACKENABLE then the Enable Boolean will be set to true or false depending on whether this event is configured to enable or disable the current track.

```
D3DXEVENTHANDLE GetUpcomingTrackEvent( UINT Track,
                                       D3DXEVENTHANDLE hEvent );
```

This function allows us to pass in an event handle to get back the next event that is due to be executed from the current event (not including the event passed in) for the specified track. This function actually searches the global timeline from the event passed in to find the next event that is scheduled. When an event is found, not only is its handle returned, but it is also cached in the controller. While this does not alter the performance or playback of events, it does allow us to subsequently call this function repeatedly, passing NULL as the event handle. Each call will step to the next event in the event queue for the specified track, until the function returns NULL. NULL indicates that there are no more events for this track. This allows us to iterate through all future events until the end of the event queue is reached.

If we pass in NULL to the initial call to this function, the handle to the next scheduled event (from the current global time) will be returned and cached for future calls.

```
D3DXEVENTHANDLE GetUpcomingPriorityBlend( D3DXEVENTHANDLE hEvent );
```

This function is the priority blend event version of the previous function. We pass in the event handle of priority blend event (or NULL to start searching from the current global time) and it will return the handle of the next scheduled priority blend event.

This function (like the previous) can be called iteratively to step through all future priority blend events that may exist. When the function returns NULL, there are no further priority blend events scheduled on the timeline and we have reached the end of the event queue.

10.9.3 Animation Sequencer Final Notes

The sequencer does not attempt to optimize or compact its event set, so do not make any assumptions about its behavior. For example, each time you pass in a value with the same global time, even if the events are identical in every way, a new event is added to the internal queue. As you can imagine, we can very quickly hit our `MaxNumEvents` limit defined for the animation controller if we are not careful. The following code would actually register 5 events with the sequencer even though they are exactly the same.

```
m_pAnimController->KeyTrackEnable ( 0 , FALSE , 5 );
m_pAnimController->KeyTrackEnable ( 0 , FALSE , 5 );
m_pAnimController->KeyTrackEnable ( 0 , FALSE , 5 );
m_pAnimController->KeyTrackEnable ( 0 , FALSE , 5 );
m_pAnimController->KeyTrackEnable ( 0 , FALSE , 5 );
```

While we would probably not do something as obvious as the above example, we do have to be very careful that we do not register events inside frequently called snippets of code. For example, a careless programmer might intend to register a single event with the animation controller, but place the code that registers it inside his main animation or render loop:

```
void UpdateAnimations ( double Time )
{
      pAnimationController->KeyTrackEnable( 0 , FALSE , 5 );
      pAnimationController->AdvanceTime ( Time );
}
```

The above code does more than just waste cycles. Every time this function is called a new event will be registered that is the same as all of the others. Our available event slots would fill up very quickly (potentially within a fraction of a second) and once the animation controller event list is full, any future calls to register additional events will fail. Therefore, you have to make sure that you set events only where and when you need them and make sure that you only set them once.

10.10 The D3DX Animation Callback Mechanism

In the DirectX 9 summer update of 2003, Microsoft added a very useful callback mechanism to their animation system. This feature allows our application to register callback events within an animation set by registering one or more callback keys. Callback keys are similar to SRT keys in that they each have a timestamp, which is specified in ticks within the period of the animation. Unlike SRT keys which are registered per-animation however, callback keys are global to all animations in the set. When the animation set is first created, these callback keys must be allocated and passed into the creation function. With each callback key, we also have the ability to pass in a pointer to a context (i.e., a pointer to any application specific data we require). When the callback event is triggered as the periodic position of the set reaches the callback key's timestamp, this context data is forwarded to an application defined callback function.

> **Note:** We have mentioned previously that the period of an animation set can be thought of as the S, R, or T keyframe within all animations with then highest timestamp (converted to seconds). As callback keys are also defined in ticks, and make up the fourth key type, they too are defined within the period of the animation set. Therefore, if we have an animation set with one animation whose last keyframe has a timestamp of 20 seconds, we would say that the period of the animation is 20 seconds. However, if we also registered a callback key at 25 seconds, the period of the animation would now be 25 seconds. It is very important that you think of callback keys as no different than SRT keys in this regard. Thus, in reality it is the scale, rotation, translation, and callback keys define the period of the animation.

An obvious case where this callback mechanism might be useful is the synchronization of sound effects with your animations. You may wish your application to play a sound effect when a certain point in the animation is reached, perhaps an explosion, for example. Using this system, your application could register a callback key with the animation set in which the explosion sound is triggered when the explosion animation sequence starts. In this case, the callback time stamp would be equal to the time (in ticks) that the animation set's explosion animation would begin. When the key is registered, we could also register some user defined data, perhaps a string containing the name of the .wav file to be played. When the event is triggered by the controller, a callback function would be executed and would be passed the name of this .wav file. Your callback function could use the passed name to actually play the sound.

So in many ways, callback keys are similar to the event keys discussed in the previous section. The main difference is the timeline used by each. As we have learned, event keys are registered on the global timeline while callback keys are registered within the local timeline of the animation set.

In our earlier discussion of animation sets, we skipped over parameters and functions that dealt with callback keys. Now we will revisit the animation set discussion and review the callback mechanism. It is important to understand exactly how the animation set deals with the callback keys behind the scenes, especially if we intend to write our own custom animation sets in the future. Therefore, we will discuss how callback keys are stored and registered with an animation set and how the animation controller gets access to that key data in order to execute the callback function.

10.10.1 Allocating Callback Keys in the Animation Set

Earlier we discussed the global function D3DXCreateKeyFramedAnimationSet, which an application can use to manually create a keyframed set of animations. Recall that we skipped the discussion of two parameters to this function; they are highlighted in bold below.

```
HRESULT WINAPI D3DXCreateKeyframedAnimationSet
(
        LPCSTR pName,
        DOUBLE TicksPerSecond,
        D3DXPLAYBACK_TYPE Playback,
        UINT NumAnimations,
        UINT NumCallbackKeys,
        CONST LPD3DXKEY_CALLBACK *pCallKeys,
        LPD3DXKEYFRAMEDANIMATIONSET *ppAnimationSet
);
```

As you may have already gathered from examining the function definition, it is at animation set creation time that we specify the maximum number of callback keys we intend to register. This allows the callback key array to be allocated when the animation set is created. Thus we see that the callback keys belong to the animation set and not to any particular animation in that set.

As the fifth parameter, we pass in the maximum number of callback keys our application might want to register with this animation set. With the sixth parameter, we are given a chance to populate this array with callback keys. While there is a separate function of the ID3DXKeyframedAnimationSet interface that allows us to store the key values in this array after the animation set has been created, often we will know the callback keys we wish to register when the set is created. So we can pass in a pointer to this application allocated callback key array at animation set creation time using this parameter. When the animation set is created, a callback key array of NumCallbackKeys will be allocated. If the sixth parameter is not NULL then this many keys will also be expected in the array passed in. The callback key data passed in is copied into the animation set's internal callback key array, so on function return, the array of D3DXKEY_CALLBACK structures can be deleted.

We now know that we allocate the memory for the callback keys at animation set creation time and can optionally pass in the callback keys at this time as well. So let us have a look at the D3DXKEY_CALLBACK structure to determine what needs to be passed into the D3DXCreateKeyFramedAnimationSet function. Keep in mind that each element in the input array represents exactly one callback key. Therefore, if we wished to register five callback keys, we would pass in a pointer to an array of five D3DXKEY_CALLBACK structures.

```
typedef struct _D3DXKEY_CALLBACK
{
    FLOAT Time;
    LPVOID pCallbackData;
} D3DXKEY_CALLBACK, *LPD3DXKEY_CALLBACK;
```

From the animation system's perspective, the only significant thing about the key is the first member of this structure. This is the timestamp of the event when we want this event to trigger a callback function. The important point to remember is that the timestamp should be specified in ticks (i.e., a periodic position) just like keyframes. As we know, this can be different from global time, especially if looping is enabled for this animation set. For example, if a looping animation set had a period of 10 seconds and we had callback key registered at 5 seconds, the callback key would be triggered at 5 seconds, and then every 10 seconds thereafter (i.e., track positions 5, 15, 25, 35, …). Once again, this is because the track time is mapped to a periodic position which is used to fetch SRT *and* callback keys that need to be executed.

> **Note**: Callback key timestamps are specified in ticks, just like keyframes. The animation set's 'TicksPerSecond' handles the conversion into seconds when needed. Callback keys are defined within the period of the animation set, just like SRT keyframes. This is important because if callback keys were registered on the global timeline (like event keys), then the callback key would not be triggered on looping animations. In our sound effect example, the sound would only play the first time though and never play again unless we manually reset the global time of the controller. Therefore, as callback events are used to allow the application to synchronize external events with animations being played, it is only logical for the callback keys and the animation data to be defined within the same timeline.

The second member of the D3DXKEY_CALLBACK structure is the context data, which will be passed to the callback function when the key is triggered.. This void pointer can be NULL or it can point to any application defined data/structure you desire. For example, let us imagine that we wanted to create an animation set which fired a callback at three different times throughout the animation. Assume the goal is to play sound effects: the first callback is triggered to play an explosion, the second is triggered to play a splashing water sound effect, and a third is triggered to play a door opening sound. We might set up the three callback keys and create the animation set like so:

```
// Somewhere in code the names of the sound effects are stored
TCHAR Sounds[3][128];
_tcscpy( Sounds[0] , "Explosion.wav" );
_tcscpy( Sounds[1] , "Splash.wav" );
_tcscpy( Sounds[2] , "DoorOpen.wav" );
..
..
..
D3DXKEY_CALLBACK keys[3];

//1st  Callback fired at a periodic position of 10 seconds
Keys[0].Time = 10.0f;
Keys[0].pCallbackData = (LPVOID)Sounds[0];

// 2nd Callback fired at a periodic position of 20 seconds
Keys[0].Time = 20.0f;
Keys[0].pCallbackData = (LPVOID)Sounds[1];

// 3rd Callback fired at a periodic position of 30 seconds
Keys[0].Time = 30.0f;
Keys[0].pCallbackData = (LPVOID)Sounds[2];

//Create an animation set called "MyAnim" which has 100 animations and has
// keyframes specified at
// 50 ticks per second. Also the set has three call-backs keys.
```

```
ID3DXKeyframedAnimationSet  *  pAnimSet = NULL;
D3DXCreateKeyFramedAnimationSet("MyAnim",50,D3DXPLAY_LOOP,100 ,3, &keys, &pAnim");
```

Two things about the previous code may seem odd when we consider how event keys work (see previous section). First, at no point do we specify the track number for which these events are to be assigned. Of course, that is because there is no need; callback keys are part of the animation set itself. Second, we are not specifying the callback *function* that is to be called when these callback events occur. As we will see in a moment, a pointer to this callback function, or to a class that contains this callback function, is actually passed into the AdvanceTime method (perhaps you remember that second parameter?). When AdvanceTime is informed by the animation set that a callback event has happened, it will retrieve the context pointer of the callback key (in our example this was a string containing the name of a sound effect) and pass it to the application-defined callback function pointer passed into AdvanceTime.

One advantage of passing the callback pointer to AdvanceTime is that we can easily change the callback function as game conditions change. The downside is that all callback keys that are triggered during a single AdvanceTime call (on all tracks) must use the same callback function. This unfortunately encourages the development of callback functions that degenerate into huge switch statements that cater for all events that may have been triggered. It might have been nice to allow the application to register the callback function pointer with the callback key itself, just like we did with the context pointer. That way, each callback could have had its own callback handler function, leading to a cleaner implementation. However, this small disadvantage is something we can certainly put up with given the overall benefits of having a callback system at all.

10.10.2 Callback Key Communication

Earlier we looked at the communication pipeline between the animation controller and its animation sets. As long as the animation set implemented the eight functions of the base class ID3DXAnimationSet, the animation controller has all it needs. The controller calls the methods of the animation set base class to perform tasks such as map the track time to a period position, fetch the SRT data for that period position, fetch an animation by index or name or fetch the entire duration/period of that animation. When we covered the ID3DXAnimationSet interface (the base class) earlier in the chapter we examined all of these methods and how they work, except one.

The one method of the base interface which must be implemented by all animation sets is called GetCallback. It is responsible for returning a callback key back to the animation controller. The method is shown next along with a list of its parameters. We will explain how this function works in some detail, because if you intend to implement your own custom animation sets, you will need to know how to implement a function of this type. When discussing this function, we will discuss the way that ID3DXKeyframedAnimationSet implements it.

> **NOTE:** How this method works may seem a little confusing at first. It will start to make more sense when we examine the way the animation controller predicts and caches the time of the next callback key. This will prevent it from having to call ID3DXAnimationSet::GetCallback for each of its animation sets in every AdvanceTime call.

The GetCallback method is a callback key querying function with a few clever tricks up its sleeve. The animation controller passes in the time where it would like the search to start and the method will return the next callback key in the animation set's timeline that is scheduled for execution in the future. What is interesting is that the animation controller passes in the search start time in track time and it expects to get back the next scheduled event in track time as well. So the GetCallback function has to take the passed track time, map it to a periodic position in the set, and then use the periodic position to start the search for the next callback key. Once the key is found, its timestamp is converted into seconds and is added to the initial track position. This is important to note. The function returns (to the animation controller) the time of the next scheduled callback event as a track position, not as a periodic position. This track position is cached by the controller and when it is reached, the callback is triggered.

For example, let us imagine that the first time AdvanceTime is called, it calls the GetCallback method of its animation set (we will assume only one track is active for now) and passes in a position of 0 seconds (i.e., the initial track position). The GetCallback method would start its key search from 0 seconds and will return the first callback key it finds after that time. If we imagine that a key was registered for 100 seconds, then the function would return a CallbackPosition of 100 along with the context data pointer that was registered with that key.

At this point the controller knows that the next callback event will not happen for another 100 seconds. Therefore, it does not have to call GetCallback for this track again until the 100 second mark has been reached. All it has to do in future calls to AdvanceTime is test the current track position against the cached next callback key time of 100 seconds. Only when the track position reaches or exceeds the currently cached callback time does work have to be done. At the 100 seconds mark, the callback function which is passed into the AdvanceTime method is executed with the callback key context (also returned from GetCallback and cached along with the key track time) passed as input. In our earlier example, this was the sound effect filename. Once the callback function completes execution and program flow returns back to AdvanceTime, the function will ask the animation set (via GetCallback) for the next callback key. If one exists, it will be cached just like the previous key, and the process starts over again.

Before we look at this procedure in more detail, let us take a look at the ID3DXAnimationSet::GetCallback function and talk about how it must be called by the animation controller. Remember, the purpose of this function is to take a time value as its parameter and return the next scheduled callback key, so that the animation controller can cache it.

```
HRESULT ID3DXAnimationSet::GetCallback
(
    DOUBLE Position,
    DWORD Flags,
    DOUBLE *pCallbackPosition,
    LPVOID *ppCallbackData
);
```

DOUBLE Position

This parameter will contain the initial track time (in seconds) from which we would like the next callback key returned. Usually, when the AdvanceTime method calls this function, it will not actually

pass in the current track time but will instead pass in the previously cached callback track time. If this was not the case, we would run the risk of missing callback events between calls.

For example, assume that we have a non-looping animation with callback keys registered at 10 and 12 seconds. Initially (when AdvanceTime is first called) the key at 10 seconds would be cached by the controller. Now imagine that we were running at a very slow frame rate and the next time AdvanceTime was called, 13 seconds had passed. The AdvanceTime method would detect that the current track position (13 seconds) is higher than the currently cached next event time (10 seconds) and would trigger the 10 second callback. This is exactly how it should behave up to this point. Now that the callback has been handled, the controller will now try to cache the position of the next scheduled event. However, if the current track position of 13 seconds was passed into GetCallback, we would be incorrectly informed that no callback keys remain. But this is not true as we know there is one at 12 seconds. Unfortunately, because we began our search for the next event at 13 seconds, we skipped it. So to avoid this issue, instead of passing in the actual track position to GetCallback, we pass the current cached time for the callback event we just handled (10 seconds in our case). Therefore, even though we are at 13 seconds actual track position, we begin the search for the next event at 10 seconds. This returns the next scheduled event at 12 seconds which is cached by the controller and executed in the next call to AdvanceTime.

> **Note:** Recall that callback key timestamps are specified in ticks. GetCallback will convert the passed value (in seconds) into a periodic position in ticks using the animation set's TicksPerSecond value. This allows it to locate the correct next scheduled callback key. Once the key is found, the timestamp is converted back into seconds and added to the start position (track time). This gives us a timestamp in track time instead of a time within the period of the animation set. It is this track position which is returned and then cached by the controller.

The controller does not try to play 'catch up' in the AdvanceTime function; it will only trigger one event (per track) within a given call. So at the end of the AdvanceTime call in our example, we have a current track position of 13 seconds and a cached next event time of 12 seconds. If another 13 seconds passes before AdvanceTime is called again, this would put the actual track position at 26 seconds. The AdvanceTime method would correctly detect that the current track position (26 seconds) is greater than the cached key time (12 seconds) and would execute the callback (admittedly, 14 seconds late). It would then pass the cached position (12 seconds) into the GetCallback method to begin a new search for the next scheduled event. As there were only two events in this example and looping is disabled, no further event is returned and cached. If looping were enabled, then the search would loop around and the next callback key returned would be the first one at 10 seconds. However, it would correctly detect that a loop had been performed and take this into account when returning the time of the callback as a track position.

Initially, this function can be confusing because it accepts and returns a track time, but works internally with data defined as periodic positions (just like the GetSRT method). So why does this function not simply accept a periodic position rather than a track position? The answer is very simple; if this function is passed a periodic position, it would have no idea how many loops of the animation have happened before the initial search time. Therefore, even when the callback key is found (which we remember is defined in local time), the function has no way of mapping it to a valid track position. If we imagine that a looping animation set with a period of 10 seconds has its GetCallback method called with a track position of 55 seconds, the function can correctly calculate that the animation has looped 5 times (50

seconds) and is currently in the fifth second of the fifth loop. This is very important because the function will need to be able to determine a proper offset within the period. If the animation set had a callback key at 8 seconds, we would find this key next (8 seconds) and would add it to the number of seconds this animation has been played out *in full* (50 seconds) to give as a track position of 58 seconds. Notice that we are talking about *completed* animations when we deal with looping. In this instance for example, we know that the event happens on the eighth second of a ten second animation. That should always be consistent, so we must offset from the start of a new looping cycle when we calculate the track time to trigger the event. This value would then be returned to the controller and cached. Obviously, if this function was not passed a track position and received only a periodic position, in the above example, a value of 5 (instead of 55) would have been passed. While we would have successfully located the next callback key at 8 seconds local time, we would have no way of knowing how to map this to track time. For further review, there is a much more detailed discussion of how these individual values are calculated and used, along with source code examples, in the workbook accompanying this chapter.

As discussed earlier, in nearly all cases, the actual track position is not used by the callback mechanism to perform the search for callback keys. Rather, we are jumping from one cached time to the next. This means that in extreme cases, the actual position we are passing into the GetCallback function may be quite different from the actual track position. As AdvanceTime is called many times per second and callback events are typically not set very close together for a single track, the animation system can catch up quickly even if it lags slightly behind the track position. This is true regardless of whether the time update has skipped past the times of many keys. For example, even though only a single callback is dealt with in a single call to AdvanceTime, typically AdvanceTime will be called somewhere between 20 to 60 times per second. This provides adequate time to quickly process any queued events and get back in synch.

While the animation controller usually passes the currently cached callback key time into the Position parameter of the GetCallback function, this is not always the case. The exception to the rule happens when the controller is in its callback key setup phase. In that case, there is no currently cached time or data. Indeed the cache is going to be invalid at several times throughout the controller's life. For example, the first time AdvanceTime is called there is no previously cached callback time. When this is the case, the actual current track position is passed in and the search for the next callback key is done from there. The cache is also invalidated if the controller time is reset to zero using ResetTime or if the track position is altered using SetTrackPosition. In all of these cases, the cache is considered dirty and the next time GetCallback is called by AdvanceTime to cache the next scheduled key time, the actual track position is passed in. Just to make sure this is clear and we understand why this must be the case, imagine that we had a currently cached time in the controller of 100 seconds, but then our application manually set the track position back to a time of 10 seconds. If we did not flush the cache, then the next time AdvanceTime is called, it would not trigger a callback until a track time of 100 seconds (the previously cached time before the reset) was reached. However, there maybe callback keys at 11 and 20 seconds that would be ignored. So in this example, when the track position was reset to 10 seconds, the actual track position would be passed into GetCallback the next time round. This would return the next scheduled event at 11 seconds, which is then cached by the controller. For all further calls to AdvanceTime, the previously cached time would be used again as the base position of the search.

> **NOTE:** The *actual* track position used to perform the callback key search only when the cache is invalidated via manipulation of the global or track timers.

DWORD Flags

The second parameter to the GetCallback function is also vitally important. If you intend to derive your own custom animation set from ID3DXAnimationSet, then you will need to understand these flags and perform your callback key search correctly, when they are passed in by the controller. This value can be zero, one, or a combination of the following flags defined in the D3DXCALLBACK_SEARCH_FLAGS enumerated type:

```
typedef enum _D3DXCALLBACK_SEARCH_FLAGS {
    D3DXCALLBACK_SEARCH_EXCLUDING_INITIAL_POSITION = 1,
    D3DXCALLBACK_SEARCH_BEHIND_INITIAL_POSITION = 2,
    D3DXEDT_FORCE_DWORD = 0x7fffffff
} D3DXCALLBACK_SEARCH_FLAGS;
```

D3DXCALLBACK_SEARCH_EXCLUDING_INITIAL_POSITION

This flag indicates that the controller would like the animation set to search for a callback key starting at the passed time, but not to include any key that is scheduled at that time. If you develop your own animation set, you will find that the controller will always pass this flag, except when the cache has been invalidated. The reason is fairly obvious when we consider what we already know about the normal operation of the callback cache.

For example, imagine that the previously cached callback key time was 10 seconds and that it has just been passed and processed. At this point the controller wishes to search for the next callback key, starting at 10 seconds. Of course, we know that there is already a callback key at 10 seconds since it is the one we just executed. Since the search function would logically find this key first, we would wind up getting stuck processing the same event over and over again. So by passing the D3DXCALLBACK_SEARCH_EXCLUDING_INITIAL_POSITION flag, the controller is instructing the animation set to search for a callback key starting at 10 seconds but to ignore the key at 10 seconds.

This completely solves any infinite looping problem and raises the question of why this flag is needed at all. After all, why not just make this behavior the default for the GetCallback function all the time? We always wish to exclude the initial position, do we not? Actually, no. Consider what happens when the cache has been invalidated because either the global or track timer has been manually altered or that this is the first time AdvanceTime has ever been called. You will recall that when this is the case, we actually perform the search starting from the real track position. If we have no previous information in the cache, we absolutely do want to search for the next callback key and include the current position. Imagine that we have a callback key at time 0.0 seconds and that at 10 seconds of track time we reset the position of the track to zero (thus invalidating the cache). At this point, the GetCallback function would be called and the track position passed would be 0.0 seconds. In this scenario, if we passed this flag, then the callback registered at 0.0 seconds would be incorrectly skipped. Therefore, we recognize that when the cache has been invalidated and the real track time is used to perform the next key search, this flag should not be specified. For all other times, it is.

D3DXCALLBACK_SEARCH_BEHIND_INITIAL_POSITION
This flag is used if you wish to search backwards from the currently passed position to find a previous callback key in the animation set's callback array. The controller itself will not typically pass this flag to the GetCallback method, but it can be useful for an application to use to retrieve a previously executed callback. If you have implemented your own animation controller, this feature might be very useful for playing backwards animations.

DOUBLE *pCallbackPosition
This parameter is where the controller passes in the address of a double which, on function return, should contain the timestamp of the next scheduled callback event in track time (seconds). The controller will cache this time so that it knows that no work has to be done and no callbacks have to be executed until this time has been reached. The conversion of the next scheduled key's timestamp from ticks into seconds is performed by the function before the value is returned in this parameter.

LPVOID *ppCallbackData
The controller uses this output parameter to get a pointer to the context data that was registered for the next scheduled key. In our earlier example, each key stored a sound filename in its context data area, thus it is the pointer to this string that would be returned here. The animation controller caches this callback data pointer along with the cached time of the event. When the cached event time is reached, the callback handler is called and the cached callback data (i.e., context data) is passed to the handler for processing. In our example, the callback function would be passed the string name of the sound effect and it could then play that sound.

While the callback system may seem complicated at first, it is actually quite simple. The following few lines of pseudo-code should give you a basic idea of how it works (how callback data is cached, etc.). This is not necessarily how it is implemented inside D3DX, but it should help better illustrate the expected behavior of the system.

```
void AdvanceTime( … )
{
    // Callback cache is invalid ?
    if ( bCacheInvalid )
    {
        // THIS IS THE SETUP CASE!
        CachedCallbackData = NULL;        // <-- NULL so handler is not called !!
        CachedCallbackPos  = TrackPos;    // <-- Significant !!
        bCacheInvalid      = false;       // Cache No Longer Invalid

        // Search for next scheduled callback key and cache it
        ProcessCallbacks();

    } // End if must reinitialize cache

    // Natural progression, performed every time ( for every track )
    if ( TrackPos >= CachedCallbackPos ) ProcessCallbacks();

    // All the rest happens here (timer increments and SRT data fetches)
    …
}
```

Here we see what the initial section of the AdvanceTime function might look like. This example only deals with a single track/animation set, but of course in the real D3DX controller there would be a cache for each active track.

The function first tests to see if the cache is invalid. The cache will be invalid if the timer of the track or the global timer has been altered. The cache will also be invalid if this is the first time AdvanceTime has been called with the animation set assigned to the track. If the cache is invalid then the cached callback data pointer is set to NULL and the cached callback event time is simply set to the current position of the track, as discussed earlier. One important thing to notice in the setup phase is that the cached position is set to the value of the track timer *before* the timer has been incremented. That is, the search for callback keys will be performed from the previous track position (not including the passed elapsed time). In this example, the ProcessCallbacks function is then called to search for the next scheduled event on that track and cache it (along with the event context data).

Beyond the setup phase we see a line of code that will be called with every call to AdvanceTime. It compares the current track position to the currently cached callback position. If the track position is greater, then it calls ProcessCallbacks to execute the callback function and fetch the next event in the schedule. Note that ProcessCallbacks is actually a multi-purpose function. When called from the setup phase, it simply searches for the next scheduled event. When called from the compare line that happens every time, it is called when the event has been reached and its callback needs to be handled. The ProcessCallbacks function is shown next:

```
void ProcessCallbacks()
{
   ULONG Flags = (CachedCallbackData == NULL) ? 0 :
                                 D3DXCALLBACK_SEARCH_EXCLUDING_INITIAL_POSITION;

   // Call the callback handler if anything is available
   if ( CachedCallbackData )
        CallbackHandler->HandleCallback(TrackIndex,CachedCallbackData);

   // Get the next callback (notice how we always pass in the last
   // callback pos, not track time)
   TrackSet->GetCallback( CachedCallbackPos, Flags,
                          &CachedCallbackPos, &CachedCallbackData);
}
```

First let us examine what this function will do if called from the setup phase of the AdvanceTime function. The first thing it will do is set the flags value to 0 because the cached callback data was set to NULL when the cache was invalid. The next line is skipped because the cached callback data will be NULL. The final line calls the animation set's GetCallback function. As the first parameter it passes the currently cached time. Because this was called from the setup phase inside AdvanceTime, the cached callback time is actually the current track position. We also pass in 0 as the flags parameter because, as discussed earlier, when the cache is invalid, we do not want to exclude the initial time from the key search. If there is a key scheduled for the current track position, we want to retrieve and cache it. Finally, notice that the CachedCallbackPos and CachedCallbackData are passed into the handler function. On function return they will contain the time of the next scheduled event on the track and any context data associated with it. At the end of the setup phase, we have a new cached position which is

no longer the current track timer (most of the time) and it will be the next event that needs to be executed.

This function is also called when the current track position reaches the currently cached key time, and it behaves a little differently in this scenario. When called during the natural progression of the AdvanceTime function, it means a cached callback key time has been reached and needs to be handled. We can see that in this case, cached callback data will exist and therefore the flags will be set to exclude the initial position in the next search for a future key. Because there is callback data, we know that we must have called this function previously and a cached event needs to be executed. So in the next line we execute the callback handler function (more on this in a moment). The final line is called to retrieve a new next event time for caching purposes. However, you should notice that when this function is called this time, the cached time is not the track position. The input time to GetCallback is the time of the event that was just handled. This way, even if the track position is many seconds in front, we do not skip any events between the last cached event and the current time.

> **Note:** The pseudo-code we are examining has been greatly simplified. For example, in ProcessCallbacks we are assuming that if there is no callback data defined then no handler needs to be called. But in fact, it is quite possible to register a callback key and assign it no context (i.e., set its callback data to NULL). There is no requirement that you always include context data, and indeed you might not need to send data into your own callback functions. So in truth, our pseudo-code design is slightly incorrect, but hopefully the general concept of key caching is clear.

We have seen how to register keys with an animation set and how the animation controller uses the animation set's GetCallback function to schedule upcoming events on a track. We have also looked at how each callback key can have user-defined data which can be passed to the callback function when the event is triggered. We even have some concept of how the AdvanceTime method executes the callback function. What we have yet to discuss is how we set this application defined callback function. This will be the purpose of our next section.

10.10.3 Setting the Callback Function

When we discussed the AdvanceTime function earlier in the chapter, we deliberately neglected to explain the second (optional) parameter. This parameter will now be examined as it is the means by which we inform the AdvanceTime function about the callback function that should be executed when a callback event is triggered. Recall that the AdvanceTime function definition is:

```
HRESULT AdvanceTime
(
    DOUBLE TimeDelta,
    LPD3DXANIMATIONCALLBACKHANDLER pCallbackHandler
);
```

As you can see, if a second parameter is passed into this function, it should be a pointer to a class of type ID3DXAnimationCallbackHandler. If this parameter is set to NULL then the function will not attempt to process any callback code.

If we take a look at the d3dx9anim.h header file, we will see that ID3DXAnimationCallbackHandler is an interface with a single method:

// Excerpt from d3dx9anim.h

```
#define INTERFACE ID3DXAnimationCallbackHandler

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

This interface is similar to the ID3DXAllocateHierarchy interface (Chapter Nine) in that it does not have an associated COM object. It is just an abstract base class that is exposed so that applications can derive their own classes from it. Your application will never be able to instantiate an instance of ID3DXAnimationCallbackHandler, so it must derive a class from it that implements the HandleCallback function. Just as the D3DXLoadMeshHierarchyFromX function is passed a pointer to an ID3DXAllocateHierarchy derived class (which it uses to callback into the application during hierarchy loading), an instance of an ID3DXAnimationCallbackHandler derived class can be passed into the AdvanceTime call. Its only method (HandleCallback) will be called by the AdvanceTime function when a callback event is triggered. As you can see, the function accepts the index of the track on which the callback event was triggered and a void pointer for the callback key context data.

Because you are responsible for implementing the HandleCallback function in your own derived classes, you can program your code to do anything you require. For example, let us continue building on our previous example where we registered three callback keys to play sound effects. We will begin by deriving our own class from ID3DXAnimateCallbackHandler and implement the HandleCallback method. In this example, we will call our derived class CSoundPlayer:

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

Now we will implement the HandleCallback function so that the name of the .wav file passed into the callback function (as pCallbackData) can be passed into a function that plays a sound.

```
HRESULT CSoundPlayer::HandleCallback( UINT Track, LPVOID pCallbackData )
{
    // cast the past string name back into a TCHAR pointer
    TCHAR * WavName = (TCHAR*) pCallbackData;

    // Play the sound
    PlaySound( WavName , NULL , SND FILENAME);
}
```

Here we see that the name of the .wav file (pCallbackData) is cast back to a TCHAR pointer. It is passed into the Win32 PlaySound function to load and play the sound. (To be sure, this is not the optimal way to play sounds in your games; this is meant only to serve as a simple callback example).

If you have multiple callback events registered on different tracks, there is a very good chance that not all of these callback keys will simply want to play back sound effects. In this case, your callback function would need to account for a wider range of functionality. One approach might be for your context data to point to an application defined structure instead just of a sound filename. The first parameter of this structure might be a flags member that you define to identify the event type to be processed (PlaySound, StopSound, FadeOut, etc.). Your callback function would simply cast the context data pointer back to the correct structure pointer and examine the flags to determine its course of action.

Finally, with your callback handler class written, you simply have to instantiate an instance of it before calling the AdvanceTime method:

```
void UpdateAnimation( double ElaspedTime )
{
   CSoundPlayer Callback;
   pController->AdvanceTime( ElapsedTime , &Callback );
}
```

10.10.4 Getting and Setting Callback Keys

The ID3DXKeyframedAnimationSet interface exposes four methods relating to the callback system which we have not discussed. Please note that these functions are implemented for the application's benefit only; they are not part of the base ID3DXAnimationSet interface and they will never be called by the animation controller. Thus, the controller will never expect them to be implemented in any custom animation set classes that you develop.

The first method allows the application to query the number of callback keys that are registered with the animation set:

```
UINT GetNumCallbackKeys(VOID);
```

Next, while there exists no way to add additional callback keys to the animation set once it has been created (recall that this memory is allocated at animation set creation time), a function is provided that allows the application to alter the value of a callback key if we know its index. The index is simply the number of the key in the internal callback key array. In our earlier example, we created an animation set with three callback keys; their indices will be 0, 1, and 2 respectively.

In order to set the value of a pre-existing key, we pass the index of the key we wish to change along with the new key information into the SetCallbackKey function.

```
HRESULT SetCallbackKey
(
    UINT Key,
    LPD3DXKEY_CALLBACK pCallbackKeys
);
```

We examined the D3DXKEY_CALLBACK structure earlier in the chapter, and saw that it contains a timestamp and a void pointer that is used to pass application data to the callback handler.

Just as our application can set the value of a callback key, we also have the ability to query key details. To do so, we can use the GetCallbackKeys method and pass in a valid key index and the address of a D3DXKEY_CALLBACK structure. If the index is a valid key index, the key information will be copied into the passed structure:

```
HRESULT GetCallbackKey
(
    UINT Key,
    LPD3DXKEY_CALLBACK pCallbackKeys
);
```

Finally, the application can also request a copy of all callback keys registered with the animation set. The GetCallbackKeys function must be passed a pre-allocated array of D3DXKEY_CALLBACK structures large enough to hold all key information in the animation set. Therefore, before calling this function, you will usually call GetNumCallbackKeys first so that you know how much memory is needed for this array.

```
HRESULT GetCallbackKeys
(
    LPD3DXKEY_CALLBACK pCallbackKeys
);
```

The following code shows how one might use this method to extract all callback keys from an animation set. It is assumed that the animation controller and the animation set are already valid and that the animation set contains a number of callback keys.

```
// Get number of keys in anim set
NumKeys = pAnimSet->GetNumCallbackKeys();

// Allocate array of the correct size
D3DXKEY_CALLBACK *pKeys = new D3DXKEY_CALLBACK[NumKeys];

// Fetch a copy of the keys into our array
PAnimSet->GetCallbackKeys( pKeys );

// Do something with keys here
…
…

// When finished release key array
delete [] pKeys;
```

Conclusion

We have now thoroughly covered all of the fundamentals of mesh hierarchies and animation using DirectX. While the animation controller documentation is a little sparse in the SDK, it should be clear that D3DX provides us with access to a very powerful and relatively easy to use set of animation interfaces.

Before moving on to skeletal animation and skinning in the next chapter, we recommend that you work through the source code to the lab projects accompanying this chapter. The next chapter will make use of use many of the techniques we have examined here, so it would be wise to read through the workbook and make sure that you are comfortable with these topics.

Lab Project 10.1 is an application that demonstrates DirectX animation at its simplest -- loading an animated X file and playing it back. The demo will teach us how to work with and render the frame hierarchy returned to us by the D3DXLoadMeshHierarchyFromX function, as well as how to use the animation controller to play the animation back. We will take this opportunity to build out our CActor class code from the previous chapter by adding the methods exposed by the D3DX animation system. Moving forward, our CActor class will become a very useful class -- it will wrap the loading and rendering of frame hierarchies and expose functions for loading and playing back animations that affect those hierarchies.

In Lab Project 10.2 we will delve even deeper into the D3DX animation interfaces and build ourselves a very handy tool. Given an X file with a single animation set, our new tool will allow us to break that animation set into multiple named animation sets and then resave the data back out to a new X file.

Again, it is very important that you understand everything we have covered so far in the course. You certainly need to be very comfortable with hierarchies and how they are animated before continuing on to the next lesson concerning skeletal animation. On the bright side, you should be pleased to know that most of the hard work is behind us now. Skeletal animation and skinned meshes are really not going to be much more complicated than the material we learned about here. What we will discover in the next chapter is that the frames of the hierarchy will simply be perceived as the bones of a skeleton with often a single mesh (although there can be multiple meshes) draped over the entire hierarchy. So, once you have tackled the lab projects in this chapter, you will be ready to move straight on to the next lesson.