# Workbook Twelve:
# Skinning Part II

# Introduction

This workbook marks the halfway point in this course. As such, one of the things we will endeavor to do is bring together many of the components we have developed throughout the course into a single application. In the accompanying textbook we learned how to generate trees. Trees will be just one element of what we intend to incorporate in this lesson.

In previous lessons our applications have essentially taken one of two forms; either a terrain demo or a scene demo. Now we will begin the process of merging these two concepts. We will add support to our IWF loading code for terrain entities so that a single IWF file can contain multiple pieces of terrain and interior scenes and objects. Previously, our applications have used a single terrain and as such the vertices in each sub-mesh of the terrain were defined in world space. Because we now wish to support multiple terrains that may be positioned at different locations throughout the scene, we will have to modify our terrain class (only slightly) so that it takes into account a world matrix for that terrain during rendering. The CScene class will now have to be able to store an array of CTerrain pointers and render them all instead of assuming only a single terrain exists.

We will also add loading support for tree entities so that objects of type CTreeActor will be added to the scene for each tree entity found in the IWF file. Internally, GILES™ uses the CTreeActor code that we developed in the accompanying textbook for its tree support. This means the creation properties stored inside the IWF file for a tree entity will work perfectly with our CTreeActor class and generate the same tree formation as displayed in GILES™. The loading code can simply instantiate a new CTreeActor for each entity it finds and feed in the tree creation parameters pulled straight from the IWF file.

Other entity types will also have support added to the IWF loading code. The GILES™ fog entity is another example. This is a simple entity that stores information about how the Direct3D fog parameters should be set during rendering.

By the end of this workbook our framework will be in a much nicer state to work with. We will be able to feed it IWF files that contain internal geometry, external reference entities for animated X files, multiple terrains, trees, and fog. Our framework will know how to store and render these components as a single scene.

While all of these housekeeping tasks and upgrades are important, they will not be the only ideas introduced. One of the main topics we will explore in this workbook will be in the development of a data driven animation system. This system will allow our applications to manage the setup and playback of complex animation sequences through a very user friendly interface. It will allow our applications (or any subsequent component) to communicate a command such as “Walk and Shoot’ to an animated actor, and the actor will know which animation sets should be set and activated, which ones should be deactivated, and which ones should be blended from one to another. Although this sounds like a complex system to implement, it will expose itself to the application via only two CActor methods. This will all be demonstrated in Lab Project 12.3 where we will see an animated skinned character walking around a terrain. The character will perform several animations such as, “Walk at ease”, “Walk and Shoot” and “Idle” when ordered by the application. Our “Action Animation System” (discussed later)

will be managing which animations sets should be assigned to which tracks on the animation mixer and which animation sets are currently in the process of having their contributions faded in or out.

This also means that we will finally have something nice to look at when we place our camera system into third person mode. When we developed our camera system way back in Chapter Four of Module I we were not yet equipped to add a real animated character to our application. As a result, our third person camera was forced to use a simple cube as a placeholder for where the character should be. The need for that placeholder is now gone and we are ready to attach an animated skinned character to the CPlayer object instead. The scene class will have a new function added that will be called by the application to load a character into a CActor object which will be attached to the CPlayer object. It will load the X file of the character mesh and its animation data and register any callback functions with the CPlayer object that are necessary for this loading process. This function will also load an .ACT file that will be used to populate the actor with information about the various actions the character can perform.

We will discover shortly that .ACT files are simple .ini files that we can create to feed information into our duly named Action Animation System (AAS for short). An .ACT file will contain one or more *actions*. Each action has a name (e.g., "Walk at Ease") and contains the names and properties of a number of animation sets that need to be set on the controller in order for that action to be played. An action called "Observe" for example might consist of multiple animations sets. It may wish to play an animation set that works on the legs to place them into a crouch position, another animation set for the torso that raises the character's binocular holding hands up to his face, and perhaps another animation set that tilts the head of the character to look through the eye pieces. An action is really just a collection of animation set names along with properties like their speed and weight. These actions will be loaded and stored by the actor and used by its AAS to set up the controller in response to the application requesting a certain action. The action system will take care of determining which tracks are free for a given animation set to use and how to configure that track to play the requested action correctly. All the application has do to is call the CActor::ApplyAction method and specify the name of the action it would like to play that matches the name we assign it in the .ACT file.

Goals for this lesson:

- Implement the Action Animation System to act as a middle tier between the application and the actor's underlying controller. The AAS will allow the application to launch complex animations in response to game events or user input with the minimum of work. A simple function call is all that will be needed to perform complex blending between groups multiple animation sets.
- Add a function to load and populate a CActor for use as the CPlayer's third person object.
- Unify components such as heightmap based terrains, trees, and interior scenes into a single application.
- Upgrade our loading code to support loading terrains definitions, tree definitions, and fog parameters directly from the IWF file. These definitions can be used as the construction parameters for our CTerrain and CTreeActor classes.
- Upgrade CTerrain to work in model space and use a world matrix for rendering and querying.

# Lab Projects 12.1 and 12.2: Trees

There is very little to discuss for the first two lab projects since the discussion of our tree class was pretty much completed in the textbook. Lab Projects 12.1 and 12.2 simply load a GILES™ tree entity and then use our CTreeActor class to build the tree based on the information stored in the IWF file. It is essentially just our mesh viewer with an updated ProcessEntities function that supports the loading of GILES™ tree entities.

The GILES™ tree entity has an identifier of 0x205, so we will create a #define for this value for entity ID comparisons during the loading process. The entity does not contain any geometry; it is just a collection of tree growth properties which we can feed into our CTreeActor::GenerateTree method.

**Excerpt from CScene.h**

```
#define CUSTOM_ENTITY_TREE      0x205   // Tree entity identifier
```

The GILES™ tree entity plug-in uses the CTreeActor code we developed in this lab project for its tree creation and rendering, so the properties stored in the IWF file are compatible with CTreeActor.

The tree entity data also has a flags member which describes how the tree should be generated. The flags are shown below. Once again we have created #define's for these in CScene.h to use during the loading process.

**Excerpt from CScene.h**

```
#define TREEFLAGS_INCLUDEFRONDS     0x1 // Tree actor to include fronds
#define TREEFLAGS_GENERATEANIMATION 0x2 // Tree should generate animation on load
#define TREEFLAGS_INCLUDELEAVES     0x4 // Tree should generate leaf information
```

The data area of the GILES™ tree entity is laid out in a very specific way, so we will set up a structure that mirrors its layout so that we can easily extract the values from the file into this structure. The TreeEntity structure is defined in CScene.h. The Flags member can be set to a combination of the flags listed above. The TREEFLAGS_INCLUDELEAVES flag is currently not supported by our lab project. It will be used in Module III when we revisit the subject of tree rendering.

```
typedef struct _TreeEntity
{
    ULONG       Flags;              // Tree actor flags
    char        BarkTexture[1024];  // The texture to use for the bark of the tree
    char        FrondTexture[1024]; // The texture to use for the frond geometry

    ULONG       GrowthSeed; // The random seed used to build the tree
    TreeGrowthProperties    GrowthProperties;    // The tree growth properties.
    D3DXVECTOR3             AnimationDirection;  // Animation direction vector
    float                   AnimationStrength;   // Animation 'strength'

    D3DXVECTOR3             InitialDimensions;   // Initial branch dimensions
    D3DXVECTOR3             InitialDirection;    // Initial growth direction.

} TreeEntity;
```

The members of this structure should be obvious to you. The Flags member contains whether or not the tree should have fronds created and/or animation data created for it. The next two members contain the names of the image files to use for the bark and frond textures. The Growth seed is the value that the random number generator should be set to during tree creation. As we have seen, we can pass this straight into the CTreeActor::GenerateTree function. The next member is the TreeGrowthProperties structure discussed in the textbook. Following that are the wind direction and wind strength used for generating animation. Finally we have the initial dimensions and growth direction for the root branch node.

## CScene::ProcessEntities - Updated

The CScene::ProcessEntities function is very familiar by now since it has been part of our loading framework for quite some time. It is called by the CScene::LoadSceneFromIWF function to extract any entity information our scene may be interested in supporting. It is passed a CFileIWF object, which at this point has loaded all the entity data and stored it in its m_vpEntityList STL vector. This function loops through the vector and extracts any information it wishes to use. Until now, this function has only supported the loading of light entities, reference entities, and sky box entities. We will now add support for GILES™ tree entities.

Because this function is rather large, we will snip out all the code that processes the entities we have already covered and replace this code with "……SNIP……".

```
bool CScene::ProcessEntities( const CFileIWF& File )
{
    ULONG      i, j;
    D3DLIGHT9  Light;
    USHORT     StringLength;
    UCHAR      Count;
    bool       SkyBoxBuilt = false;

    // Loop through and build our lights & references
    for ( i = 0; i < File.m_vpEntityList.size(); i++ )
    {
        // Retrieve pointer to file entity
        iwfEntity * pFileEntity = File.m_vpEntityList[i];

        // Skip if there is no data
        if ( pFileEntity->DataSize == 0 ) continue;
```

The first section of code (shown above), sets up a loop to step through every entity in the CFileIWF object's internal entity vector. Inside the loop we get a pointer to the current entity to be processed (as an iwfEntity structure). We then read the entity's data size and if we find it is zero, we skip it. An entity with a data size of zero is probably one that has been erroneously written to the file.

In the next section of code we test to see if the current entity is a light entity . If it is and we have not already loaded the maximum number of lights, we add a light to the scene. The code that adds the light to the scene's light array has been snipped since this code has been with us for a very long time.

```
        //skip all lights if we've already reached our limit, or maximum
        if ( (m_nLightCount < m_nLightLimit && m_nLightCount < MAX_LIGHTS)
              && pFileEntity->EntityTypeMatches( ENTITY_LIGHT ) )
        {
            …… SNIP ……

        } // End if light
```

The next code block is executed if the current entity has an author ID of 'GILES'. If it does then we know this is either a sky box entity, a reference entity, or a tree entity; all of which we wish to support in our loading code. When this is the case we set up some entity structures that will be used to extract the information into. Notice that the new addition here is the instantiation of a TreeEntity structure. We then get a pointer to the entity's data area.

```
        else if ( pFileEntity->EntityAuthorMatches( 5, AuthorID ) )
        {
            SkyBoxEntity SkyBox;
            ZeroMemory( &SkyBox, sizeof(SkyBoxEntity) );

            ReferenceEntity Reference;
            ZeroMemory( &Reference, sizeof(ReferenceEntity ));

            TreeEntity Tree;
            ZeroMemory( &Tree, sizeof(TreeEntity));

            // Retrieve data area
            UCHAR * pEntityData = pFileEntity->DataArea;
```

Now that we have a pointer to the entity data area we will test the entity ID to see what type of GILES™ entity it is. We set up a switch statement to handle skybox, reference, and tree entity types. As we have already covered the code for the first two in previous lab projects, the code has been snipped here.

```
            switch ( pFileEntity->EntityTypeID )
            {
                case CUSTOM_ENTITY_REFERENCE:

                    …… SNIP ……

                    break;

                case CUSTOM_ENTITY_SKYBOX:

                    …… SNIP ……

                    break;
```

In the next section of code we see the case that is executed when the entity type is a GILES™ tree entity. Extracting the values from the data area is simply a case of copying the correct number of bytes from the entity data pointer (retrieved above) into our TreeEntity structure for each property, and then advancing this pointer so that it points at the next value to be read. In the section shown below we

extract the flags ULONG and the string containing the bark texture filename. Notice how we are copying each value into their corresponding members in the TreeEntity structure.

```
                case CUSTOM_ENTITY_TREE:

                    // Retrieve the fog entity details.
                    memcpy( &Tree.Flags, pEntityData, sizeof(ULONG) );
                    pEntityData += sizeof(ULONG);

                    // Bark Texture
                    memcpy( &StringLength, pEntityData, sizeof(USHORT) );
                    pEntityData += sizeof(USHORT);

                    if ( StringLength > 0 )
                    memcpy( Tree.BarkTexture, pEntityData, StringLength );
                    pEntityData += StringLength;
```

The next value in the entity data area of the tree is a value describing how many frond textures our tree uses. We currently only support a single frond texture so we skip all but the first texture listed (feel free to experiment with this setting as you please). We then extract the first texture name into the TreeEntity structure.

```
                    // Frond Texture Count
                    Count = *pEntityData++;

                    // Loop through textures in the file
                    for ( j = 0; j < (ULONG)Count; ++j )
                    {
                        // Get string length
                        memcpy( &StringLength, pEntityData, sizeof(USHORT) );
                        pEntityData += sizeof(USHORT);

                        // skip all others stored.
                        if ( j < 1 )
                        {
                            // Read String
                            if ( StringLength > 0 )
                            memcpy( Tree.FrondTexture, pEntityData,StringLength );
                            pEntityData += StringLength;

                        } // End if first frond texture
                        else
                        {
                            // Skip String
                            pEntityData += StringLength;

                        } // End if unsupported extras

                    } // Next Frond Texture
```

Notice that in the frond texture name loop, we skip all but the first texture found. For all others we still increment the data pointer to move it past any other frond texture names that may be stored there. At the

end of this loop, regardless how many frond texture names were stored, our data pointer will be pointing at the next property to be extracted.

The GILES™ tree entity also supports a concept of separate leaves (in addition to the fronds), but our class does not currently support this technique (Module III will address this). Therefore, we will simply retrieve the number of leaves stored in the file and will then increment the data pointer past the string stored for each leaf.

```
                    // Leaf Texture Count
                    Count = *pEntityData++;

                    // Loop through leaf textures in the file
                    for ( j = 0; j < (ULONG)Count; ++j )
                    {
                        // Get string length
                        memcpy( &StringLength, pEntityData, sizeof(USHORT) );
                        pEntityData += sizeof(USHORT);

                        // Leaves are not yet supported, skip all texture names.
                        pEntityData += StringLength;

                    } // Next Leaf Texture
```

Next we extract the random generator seed value of the tree into the tree entity structure.

```
// Growth Seed
memcpy( &Tree.GrowthSeed, pEntityData, sizeof(ULONG) );
pEntityData += sizeof(ULONG);
```

The next block of data stored in the IWF file is basically a mirror of the TreeGrowthProperties structure. While the following block of code looks rather complex, all it is doing is fetching the values for each property in order and copying them into the TreeGrowthProperties member of the TreeEntity structure.

```
// Growth Properties
memcpy( &Tree.GrowthProperties.Max_Iteration_Count,pEntityData, sizeof(USHORT) );
pEntityData += sizeof(USHORT);

memcpy( &Tree.GrowthProperties.Initial_Branch_Count,pEntityData, sizeof(USHORT) );
pEntityData += sizeof(USHORT);

memcpy( &Tree.GrowthProperties.Min_Split_Iteration,pEntityData, sizeof(USHORT) );
pEntityData += sizeof(USHORT);

memcpy( &Tree.GrowthProperties.Max_Split_Iteration,pEntityData, sizeof(USHORT) );
pEntityData += sizeof(USHORT);

memcpy( &Tree.GrowthProperties.Min_Split_Size, pEntityData, sizeof(float) );
pEntityData += sizeof(float);

memcpy( &Tree.GrowthProperties.Max_Split_Size, pEntityData, sizeof(float) );
pEntityData += sizeof(float);
```

```
memcpy( &Tree.GrowthProperties.Min_Frond_Create_Iteration,
        pEntityData, sizeof(USHORT) );

pEntityData += sizeof(USHORT);

memcpy( &Tree.GrowthProperties.Frond_Create_Chance, pEntityData, sizeof(float) );
pEntityData += sizeof(float);

memcpy( &Tree.GrowthProperties.Frond_Min_Size, pEntityData, sizeof(D3DXVECTOR3) );
pEntityData += sizeof(D3DXVECTOR3);

memcpy( &Tree.GrowthProperties.Frond_Max_Size, pEntityData, sizeof(D3DXVECTOR3) );
pEntityData += sizeof(D3DXVECTOR3);

memcpy( &Tree.GrowthProperties.Two_Split_Chance, pEntityData, sizeof(float) );
pEntityData += sizeof(float);

memcpy( &Tree.GrowthProperties.Three_Split_Chance, pEntityData, sizeof(float) );
pEntityData += sizeof(float);

memcpy( &Tree.GrowthProperties.Four_Split_Chance, pEntityData, sizeof(float) );
pEntityData += sizeof(float);

memcpy( &Tree.GrowthProperties.Split_End_Chance, pEntityData, sizeof(float) );
pEntityData += sizeof(float);

memcpy( &Tree.GrowthProperties.Segment_Deviation_Chance,
       pEntityData, sizeof(float) );
pEntityData += sizeof(float);

memcpy( &Tree.GrowthProperties.Segment_Deviation_Min_Cone,
        pEntityData, sizeof(float) );
pEntityData += sizeof(float);

memcpy( &Tree.GrowthProperties.Segment_Deviation_Max_Cone,
       pEntityData, sizeof(float) );
pEntityData += sizeof(float);

memcpy( &Tree.GrowthProperties.Segment_Deviation_Rotate,
        pEntityData, sizeof(float) );
pEntityData += sizeof(float);

memcpy( &Tree.GrowthProperties.Length_Falloff_Scale, pEntityData, sizeof(float) );
pEntityData += sizeof(float);

memcpy( &Tree.GrowthProperties.Split_Deviation_Min_Cone,
        pEntityData, sizeof(float) );
pEntityData += sizeof(float);

memcpy( &Tree.GrowthProperties.Split_Deviation_Max_Cone,
       pEntityData, sizeof(float) );
pEntityData += sizeof(float);

memcpy(&Tree.GrowthProperties.Split_Deviation_Rotate, pEntityData,sizeof(float) );
pEntityData += sizeof(float);
```

```
memcpy( &Tree.GrowthProperties.SegDev_Parent_Weight, pEntityData, sizeof(float) );
pEntityData += sizeof(float);

memcpy(&Tree.GrowthProperties.SegDev_GrowthDir_Weight,pEntityData,sizeof(float) );
pEntityData += sizeof(float);

memcpy( &Tree.GrowthProperties.Branch_Resolution, pEntityData, sizeof(USHORT) );
pEntityData += sizeof(USHORT);

memcpy( &Tree.GrowthProperties.Bone_Resolution, pEntityData, sizeof(USHORT) );
pEntityData += sizeof(USHORT);

memcpy( &Tree.GrowthProperties.Texture_Scale_U, pEntityData, sizeof(float) );
pEntityData += sizeof(float);

memcpy( &Tree.GrowthProperties.Texture_Scale_V, pEntityData, sizeof(float) );
pEntityData += sizeof(float);

memcpy( &Tree.GrowthProperties.Growth_Dir, pEntityData, sizeof(D3DXVECTOR3) );
pEntityData += sizeof(D3DXVECTOR3);
```

After the TreeGrowthProperties structure we have the animation wind direction vector and the wind strength, so let us read these values next (a 3D vector and a float)

```
// Animation Properties
memcpy( &Tree.AnimationDirection, pEntityData, sizeof(D3DXVECTOR3) );
pEntityData += sizeof(D3DXVECTOR3);

memcpy( &Tree.AnimationStrength, pEntityData, sizeof(float) );
pEntityData += sizeof(float);
```

Finally, we read in the 3D vector describing the initial dimensions of the root node of the tree and its growth direction (also a 3D vector).

```
// 'Initial' values
memcpy( &Tree.InitialDimensions, pEntityData, sizeof(D3DXVECTOR3) );
pEntityData += sizeof(D3DXVECTOR3);

memcpy( &Tree.InitialDirection, pEntityData, sizeof(D3DXVECTOR3) );
pEntityData += sizeof(D3DXVECTOR3);
```

We have now loaded all the data from the entity into our TreeEntity structure. Next we test to see if the flags we extracted have the TREEFLAGS_INCLUDEFRONDS bit set. If so, we set the Include_Fronds Boolean of the TreeGrowthProperties structure to true so that when we feed it into the CTreeActor::GenerateTree function, it will know to generate fronds for the tree.

```
// Setup data from flags
if ( Tree.Flags & TREEFLAGS_INCLUDEFRONDS )
     Tree.GrowthProperties.Include_Fronds = true;
```

At this point the TreeGrowthProperties structure stores all the tree information we need, so let us generate the tree. This task is actually handed off to the CScene::ProcessTree method. This is a new method that we have added to this class and will discuss next.

```
// Process the tree entity (it requires validation etc)
if ( !ProcessTree( Tree, (D3DXMATRIX&)pFileEntity->ObjectMatrix ) ) return false;

break;

} // End Entity Type Switch
```

The ProcessTree method takes two parameters. For the first parameter we pass the TreeEntity structure that we have just populated with the data from the entity and as the second parameter we pass in the entity's world matrix (every entity has a world matrix member describing its location in the scene).

```
        } // End if custom entities

    } // Next Entity

    // Success!
    return true;
}
```

With the changes to the ProcessEntities function covered let us now look at the new CScene::ProcessTree function. This function has the task of creating a new CTreeActor using the values stored in the passed TreeEntity structure and adding it to the scene's CObject array,

## CScene::ProcessTree

The first section of this function (shown below) creates a new CTreeActor and registers a scene attribute callback function with it. It then sets the tree actor's branch and frond material to the filenames stored in the passed TreeEntity structure and calls the CTreeActor::GenerateTree method to build the tree meshes.

```
bool CScene::ProcessTree( const TreeEntity& Tree, const D3DXMATRIX & mtxWorld )
{
    HRESULT       hRet;
    CTreeActor  * pTreeActor  = NULL;
    CObject     * pTreeObject = NULL;

    // Allocate a new tree actor
    pTreeActor = new CTreeActor;
    if (!pTreeActor) return false;

    // Setup and generate the tree
    pTreeActor->RegisterCallback( CActor::CALLBACK_ATTRIBUTEID,
                                  CollectAttributeID, this );

    pTreeActor->SetBranchMaterial( Tree.BarkTexture, NULL );
```

```
    pTreeActor->SetFrondMaterial( Tree.FrondTexture, NULL );

    pTreeActor->SetGrowthProperties( Tree.GrowthProperties );

    hRet = pTreeActor->GenerateTree( D3DXMESH_MANAGED,
                                     m_pD3DDevice,
                                     Tree.InitialDimensions,
                                     Tree.InitialDirection,
                                     Tree.GrowthSeed );

    if ( FAILED(hRet) ) { delete pTreeActor; return false; }
```

At this point the tree skins have been built so we will check the flags member of the passed TreeEntity structure to see if animation should be generated for this tree. It so, we pass the wind direction and the wind strength properties into the CTreeActor::GenerateAnimation method.

```
    // Generate the animation if requested
    if ( Tree.Flags & TREEFLAGS_GENERATEANIMATION )
    {
        hRet = pTreeActor->GenerateAnimation( Tree.AnimationDirection,
                                              Tree.AnimationStrength );
        if ( FAILED(hRet) ) { delete pTreeActor; return false; }

    } // End if generate animation
```

With our CTreeActor now completely built, let us make room at the end of the scene's CActor array to store its pointer.

```
    // Store this new Actor
    if ( AddActor( ) < 0 ) { delete pTreeActor; return false; }
    m_pActor[ m_nActorCount - 1 ] = pTreeActor;
```

With the tree actor's pointer added to the scene's actor array, we store the actor and its world matrix in a newly allocated CObject before resizing the scenes CObject array and storing CObject's pointer in it.

```
    // Now build an object for this TreeActor (standard identity)
    pTreeObject = new CObject( pTreeActor );
    if ( !pTreeObject ) return false;

    // Copy over the specified matrix
    pTreeObject->m_mtxWorld = mtxWorld;

    // Store this object
    if ( AddObject() < 0 ) { delete pTreeObject; return false; }
    m_pObject[ m_nObjectCount - 1 ] = pTreeObject;

    // Success!!
    return true;
}
```

That completes the framework changes for Lab Project 12.2. Our framework now has support for loading GILES™ tree entities and generating skinned actors for them.

# Lab Project 12.3

Lab Project 12.3 is the first project that will employ our Action Animation System. It will be implemented into the CActor interface and used by the application in this lab project to apply animated sequences to the character as he navigates the level. In this project we will also update our framework and loading code to add IWF file terrain and fog support. The first order of discussion in this section will be the design and implementation of the Action Animation System.

## The Action Animation System

The goal of the action system we will develop is not to provide the most flexible animation mixing system imaginable. Indeed such a system would be quite complex and hard to use as a teaching exercise. Instead, our goal is to provide a simple way to describe how the various animation controller tracks should be set up for a specific action, while providing a relatively easy to use interface for the end user. Let us be clear about what we mean by an action.

## What is an Action?

In our system, we assign the word ‘action’ to the collection of animation sets, and their associated properties, that must be set on the animation controller for a high level action to be played. It is quite common for an artist to create animation sets for a given model such that they are localized to various regions of the body. As an example, a character may have three animation sets created for it that animate only the legs of the character in a certain way. One animation set might be called ‘Legs Idle’ and simply shuffles the legs from side to side while the character is awaiting further instruction. A second animation set might be called ‘Legs Walk’ and needs to be played when the character is walking. A third animation set might be called ‘Legs Run’ and when played, launches the legs of the character into a full sprint. Remember, these animation sets affect only the legs. Of course, the artist would also need to define animation sets for the torso. For example, there may be two animation sets called ‘Torso Ready’ and ‘Torso Shooting’. When the ‘Torso Ready’ animation set is played, the character should hold his weapon in his hands ready for action but not yet taking an aggressive posture. We might think of this as being like the arms of a soldier as he walks on patrol with his assault rifle. The animation set ‘Torso Shooting’ could really be as simple as a single keyframe set that when played, poses the arms of the soldier up near his chest as if the soldier is firing a weapon. Of course, animation sets may exist for other parts of the body too; there may be animation sets that tilt the head of the soldier to one side when in the firing position such that he is looking through the eye piece of his weapon. For now, we will just go forward with the arms and legs example for ease of explanation.

Although the artist could create animation sets that animate both the torso and the legs, the benefits of the segmented approach is that it relieves the artist from having to create animation sets for every combination of torso and leg animations that the application may wish to play (although to be sure, many game development shops will do exactly that). Imagine that the artist created a single animation set called ‘Walk and Shoot’ that animated both the legs in a walking motion and the arms in a shooting

motion. The artist would also have to create such animation sets for every combination. For example, he may also have to create the following animation sets "Run and Shoot", "Idle and Shoot", Walk and Ready", "Run and Ready", "Idle and Ready". When you imagine the various animation sequences which can be applied to the legs alone "Jump, Crouch, and Kick", the artist would then have to create 'Ready' and 'Shoot' versions of these animations too. As you might assume, this would put a heavy asset creation burden on the artist.

By using a segmented approach as discussed above, the artist can create isolated animations for the various parts of the character's body. These separate components can then be mixed and matched by our engine at run time. The artist would create only three animation sets for the legs in this example. 'Legs Idle', 'Legs Walk', and 'Legs Run'. If we imagine that the character is running and then wants to fire his weapon, we can still leave the 'Legs Run' animation set playing but then also play the 'Shooting' animation set. Since these two sets, which are playing simultaneously, work on totally separate parts of the body, they do not conflict with each other. If the character is still firing his weapon but then would like to start walking instead, we can just swap out the 'Legs Run' animation for the 'Legs Walk' animation. The legs would now play a different sequence while the torso continues to fire the weapon, unhindered by the change applied to the legs.

Although the segmented animation set design is often desirable, it does shift the burden to the application with respect to handling which animation sets should be played simultaneously to perform a given action. For example, we know that in the previous example, if we wanted to have our character perform an action such as 'Run and Fire', we would have to assign both the 'Legs Run' and 'Shooting' animation sets to active tracks on the animation mixer. This is one of the core ideas that our Action Animation System addresses. The important thing at the moment is for us to understand that an action can be thought of quite abstractly as an action to be performed by the object. For instance, we may want our actor to swim, jump, run, and walk. But each of these actions may need to be comprised of many different animation sets under different circumstances. An action (in the terms outlined by this system) is that collection of animation sets (and their properties) which describes to the animation controller which components are required to carry out the act of, for instance, swimming or walking.

By integrating this system directly into our Actor class, it allows us to use a simple data driven approach for setting up the controller, without having to necessarily work through separate components. This keeps the complexity of the system low from a user standpoint. Because are not yet ready to employ a full scripting system (we will introduce you to that in Module III), we currently make use of a static descriptor file format. Because of the static nature of the description components, we will be bound to a 'pull only' process (i.e. it cannot change based on the circumstances at any given time during game-play). These descriptor files are essentially just standard Windows™ *'.ini'* files, which we process by making use of the standard Win32 API functions `::GetPrivateProfileInt()`, and `::GetPrivateProfileString()`. We will discuss the format of these files in a moment and examine their contents. An Action file will have an '.act' extension and contain some number of actions. Each action will have a name that the application will use to apply that action to the character (e.g., "Walk at Ease" or "Run and Shoot"). Each action in the .act file will contain a list of the names of animation sets which need to be played simultaneously to achieve that action as well as the properties that should be assigned to the tracks for those animation sets. This means we can also control the speed and contribution strength of a given animation set within an action when it is applied.

Although the ability to store a list of animation sets and their properties with an assigned action name would be a convenient way for the application to specify the various animation sets that should be set on the controller to achieve a given action, our action system provides a powerful blending feature that will allow it to intelligently make decisions about which tracks on the mixer should be used. It will also determine which animation sets from a previously played action should be slowly diminished as we phase in an animation set from a newly applied action. The ability for us to be able to blend the animation sets from a previous action into the animation sets of a new action over time is an essential component of our system.

## Action Blending

To understand why blending is useful (and often necessary) let us continue with our previous example of the two animation sets defined for the torso of the character; 'Ready' and 'Shooting'. We will also assume for the sake of this discussion that both these animation sets contain just a single keyframe. That is, they do not really animate the torso; they just contain a snapshot of the character's arms in a different position. When the 'Ready' animation set is playing the arm frame matrices are set such that the soldier is holding his weapon in front of his chest. The 'Shooting' animation set is similarly a single keyframe looping set that when played, positions the arms of the character such that he is holding his weapon up to his face.

Figure 12.2 shows the state of the torso when each animation set is being played. Let us now imagine that the character's torso is currently in the 'Ready' pose and we wish it to assume the 'Shooting' pose when the user presses the fire button. The wrong way to handle this, when the fire button is pressed, would be to remove the 'Ready' pose from its track on the animation mixer and assign the 'Shooting' pose in its place. The next time the scene is rendered, the arms of the character will be in the shooting pose as requested, but this is a terrible way to accomplish that objective.

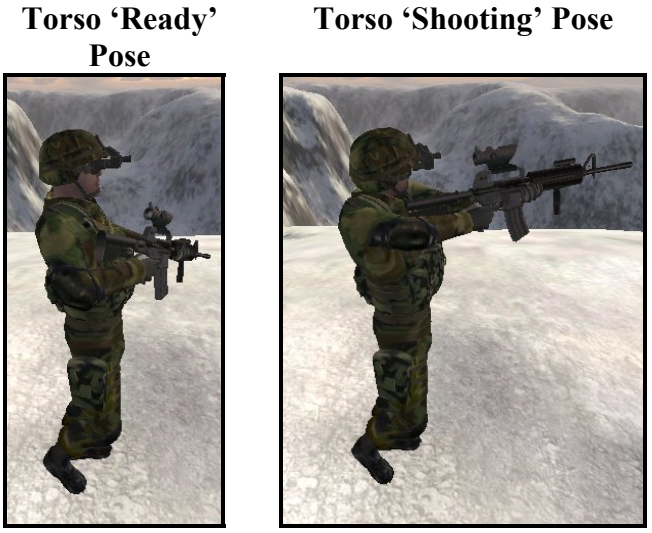


**Figure 12.2**

The obvious problem with this approach of course is the lack of any smooth transition between states. Simply swapping one pose for another would mean that the arms of the character would abruptly change from the ready position to the shooting position within the time of a single frame update.

In reality, we would expect to see the character's arms slowly transition from the ready pose to the shooting pose. But if we do not employ any blending support at the code level, the artist would have to make transition animations. When we consider the sheer number of animations an artist would need to make for each pose for each section of the character's body, and then consider that the artist would then need to build transition animation sets from each pose to every other pose, the asset demands on the

artist for just a single character would be staggering. By exposing run-time blending support in our action system, we can get those transitions for free. So let us now discuss how our system may provide asset-free transition sequences.

When we think about the problem in Figure 12.2 and factor in the functionality that is exposed by the D3DX animation system, a solution arises that makes use of sequencer events and the weight properties of mixer tracks.

Each action in the .act file will contain a number of animation set definitions. A *set definition* is comprised of the name of one of the animation sets defined in the X file along with properties that will be applied to the mixer track when the animation set is played. One property that we will also introduce is the ability to assign an animation set a 'Group' name. Within a given action, there should never be more than one animation set with the same group name, but animation sets contained in different actions can belong to the same group. For example, we might have two actions defined in our .act file. In each action there might be an animation set definition that describes how the torso of the character should be animated. (It is important to note that only one action can be applied to the character at any one time.) We will imagine that the animation set in the first action that animates the torso is called 'Ready' and the torso animation set being used by the second action is called 'Shooting'. Because both of these animation sets exist in different actions, but animate the same region of the character, we will assign them the same group name (e.g., 'Arms').

Now let us imagine that our application applies the first action. The 'Ready' animation set will be assigned to the first empty track on the mixer to animate the arms of the character in the ready pose. We will cache some information in the actor at this point that says that the animation set currently assigned to that track belonged to a group with the name 'Arms'. So far, nothing out of the ordinary is happening until the user presses the fire button and the second action is applied. We know that the second action contains as one of its animation sets the 'Shooting' animation set, which also belongs to a group with the name 'Arms'. Here is where things get interesting. Rather than just replacing all the animation sets that were set on the controller when the previous action was applied with the animation sets defined in this new action, we will do the following for any set that has been configured for blending in the action file:

- For each animation set definition in the action we are about to apply we will search for a track on the controller that is not in use (i.e., a free track).
- If the animation set has been configured to blend in the .act file, it will also have a property assigned to it called Mix Length. This is a value describing the number of seconds we would like the animation set's weight to rise from zero to full strength. For example, a value of 5 seconds means that we would like this animation set to slowly fade in to full contribution over five seconds.
- At this point we register a sequencer event with the new track. The event is a track weight event that describes our fade in from zero to full strength with a duration of five seconds, starting immediately.
- Since only one animation set from a given group should be current at one time, we deactivate any animation sets that belong to the same group as the one we have just activated. In our example, we have just set the 'Shooting' animation set on a mixer track. Since this set belongs to the 'Arms' group, we should now deactivate any animation sets assigned by a previous action

that belongs to the same group. In our example, the 'Ready' animation set would currently be in use by the mixer and is part of the 'Arms' group.

- Since the 'Shooting' set was configured to blend in over a period of five seconds, we will do the opposite for any animation set that belongs to the same group that was applied by a previous action. That is, we will set a sequencer event that will fade the weight/contribution of the 'Ready' set from its current full strength setting to zero over a period of five seconds.
- We will also set a second sequencer event that will be triggered in 5 seconds time (when the set has fully faded out) to disable the track with the 'Ready' set.
- We will cache the information somewhere that this track is now fading out so that it will not be used by any future actions we apply until the fade out is complete.
- When any tracks have their enabled state set to false by the sequencer event, we know that the set that was applied to that track from a previous action has fully faded out and that it can now be used by the system to apply the sets of future actions too (i.e., it becomes a free track).

Figure 12.3 shows the basic 'blending' transition that will occur when a set is applied to the animation controller which shares the same group name with an animation set already set on the mixer. We are showing the blending case here, but as you will see, blending is an optional property we can specify for any animation set in the .act file.

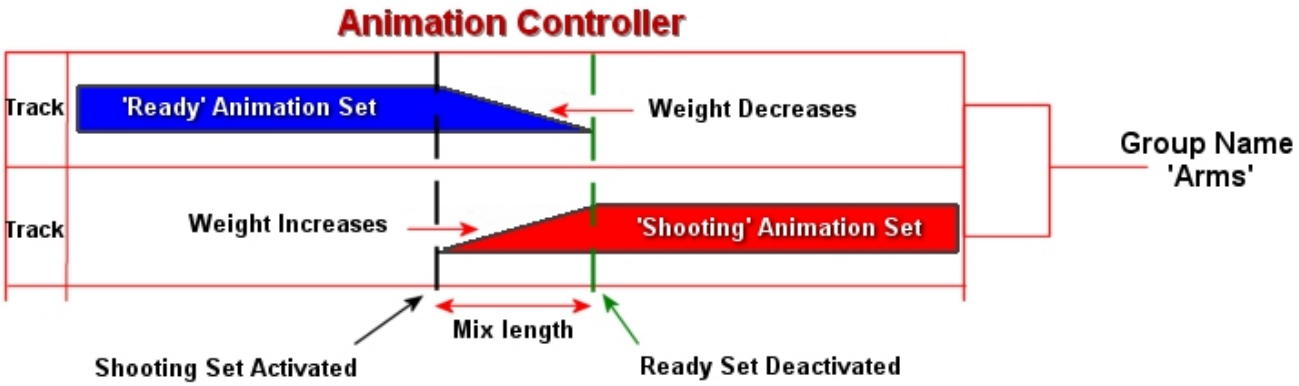


**Figure 12.3**

In Figure 12.3 it is assumed that the 'Ready' animation set and the 'Shooting' animation set belong to two different actions but also belong to the same group ('Arms'). It is also assumed that the action that was applied first to the actor was the one that contained the 'Ready' animation.

In this diagram we show the process that occurs between two animation sets from two different actions when the second action is applied and both animation sets share the same group name. We see that halfway through the playing of the 'Ready' animation set a new action is applied which contains the 'Shooting' animation. This animation is part of the same group. In the new action we are about to apply, the 'Shooting' animation set is configured to blend in over a period of time specified by its Mix Length. The 'Shooting' animation set is assigned to a mixer track and a sequencer event is set immediately that will fade that track's weight from 0.0 to its full weight over the time specified in the Mix Length property. If we assume for the sake of this discussion that the Mix Length property assigned to the shooting animation set is 5 seconds, the contribution of the 'Shooting' animation set on the actor's bone

hierarchy will become stronger over a 5 second period. Furthermore, at this exact time, we also set a sequencer event for any previous tracks which belong to the same group to fade out over the same period of time. In this simple example, the 'Ready' animation set belongs to the same group, so it has a sequencer event set for it that will fade its weight from its current value to zero over the next 5 seconds. We will also schedule a sequencer event on the 'Ready' track that will be triggered in 5 seconds time to disable the track when the fade out is complete. At this point, we will make a note that the track is ready to be used again.

Although only one action can be applied to an actor at any given time and no single action can contain more than one animation set with the same group name, multiple sets belonging to the same group can all be contributing to the hierarchy simultaneously during these transition phases. If you look at the area between the markers (the vertical dashed lines) in Figure 12.3, you will see that while the action that contains the 'Shooting' set is considered to be the current action in use by the system, animation sets from the previous action are fading out over time. We can see that during the Mix Length period, two animation sets (one from the current action and one from the previous action) are being used to animate the bone hierarchy simultaneously.

At a point halfway through the Mix Length period, the two sets would have an equal contribution to the hierarchy. We can think of this as positioning the gun halfway between the ready pose and the shooting pose (see Figure 12.4). Therefore, as the weight of the 'Ready' pose diminishes and the weight of the 'Shooting' pose increases, the arms of our actor will slowly move from the ready pose to the shooting pose and we have a transition animation without the artist ever having to create a custom one. You might also imagine that if you applied several actions to the actor over a very short period of time, which each had fading animation sets belonging to the 'Arms' group, after the last action had been applied, you would have one animation set for that group fading in, and multiple animation sets (from the various previous actions) in various stages of fading out. Finally, you can couple that with the fact that multiple actions can contain multiple animation sets, each belonging to multiple shared groups, and we have a lot of flexibility here (and a lot of work to do). Of course, we will have to make sure that we have sufficient spare mixer tracks to handle all the transitioning out we will need to perform when new actions are applied and older actions are still fading out.

This same system can be used to transition between animation sets belonging to the 'Legs' group as well. When an action is applied to animate the legs of the character in a running motion, this set could be faded in over a period of time. If the animation set for the 'Legs' group from a previous action was a walking sequence, the legs would transition from a walk to a run over a period of time, instead of instantly snapping from one to the other; very nice indeed.

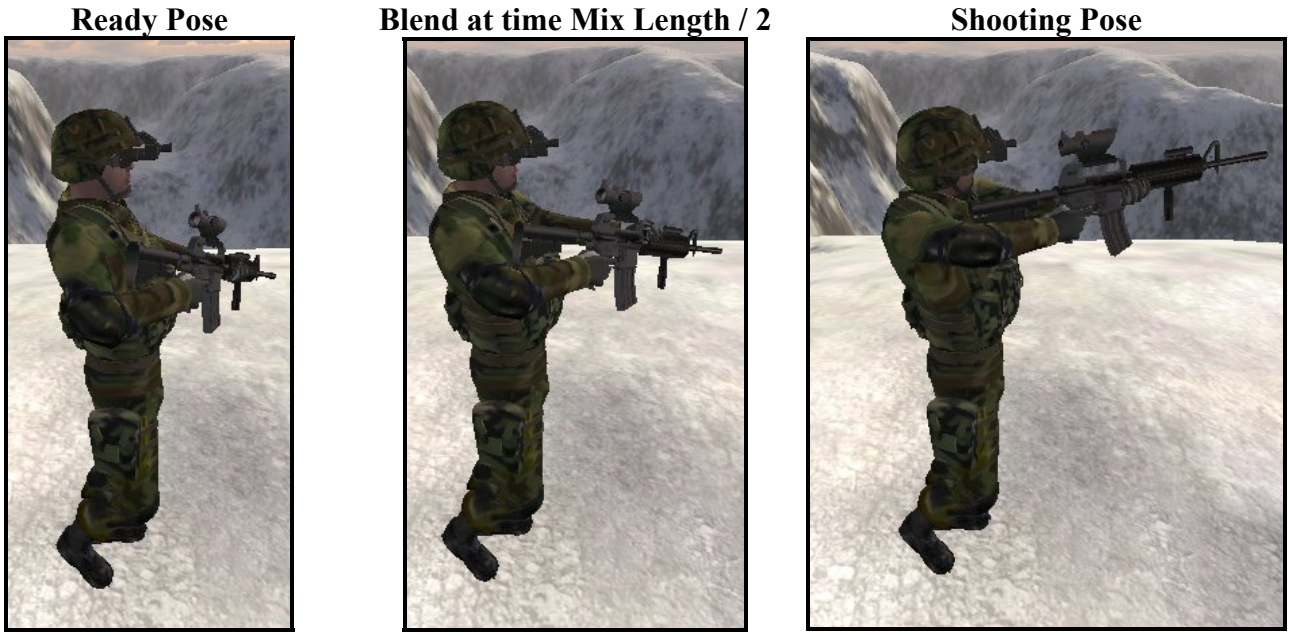


**Figure 12.4 : Transition through animation set blending**

We will need a way to synchronize the timers for the various tracks when applying actions. For example, if we wish to play the 'Running' animation on the legs and we are currently playing a 'Walking' animation, we would not want the running animation to start from the beginning each time. If the walking animation was halfway through when the running animation set was activated, the two clocks would be wildly out of sync. We would definitely not want the left leg of the character halfway through its walk pose to snap back to its starting position in the run pose. Therefore, in our .act file we will also be able to specify how the clock should be set when an animation set is applied to a mixer track. For example, when applying the running animation to the legs, we would like to set the track position to that of the walk animation set's track that is about to fade out. In other words, we want the new track to take its time from a currently playing track from the same group. That way, the running animation will gracefully take over from the walking animation over the Mix Length period.

As you will see however, all of this functionality is optional. We can have sets defined in actions that are not set to blend or transition. When such animation sets need to be played, any animation sets currently assigned to the controller that share the same group name will be deactivated instantly and the new animation set will take its place. We will also have the option not to synchronize the track clock of a newly assigned animation set to that of the group to which it belongs and may wish to simply play it from the beginning.

Before we get into the discussion of the implementation details and start looking at the contents of an .act file, we have assembled a small glossary of terms that we have collected so far so that there is no confusion as we move forward.

# The Action System – Glossary of Terms

**ACT File**
This is the primary descriptor file which contains the information required to set up the controller tracks for playing out the specified actions. It contains the definitions of one or more actions. Since these are simple text files, we can create them in any text-base editor (e.g., Notepad).

**Action**
Some combination of animations that convey a higher level animation concept we wish to perform. For example, we may want our actor to swim, jump, run and walk. Each of these actions may need to be comprised of many different animation sets under different circumstances. An action (in the terms outlined by this system) is that collection of animation set names and their properties which describe the components required to carry out the higher level performance.

Every action has a name (e.g., "Walk and Shoot"). It is this name that our application will pass to the CActor::ApplyAction method in order to configure the controller to play that action. The action is comprised of one or more set definitions.

**Set Definition**
Each action stores one or more set definitions. An individual set definition describes how we would like a single animation set (one set out of a possible many required to perform an action) to be processed and initialized when it is applied to an animation controller mixer track. The various properties stored within a set definition are items such as the name of the animation set to apply, the speed we would like it to play, how much it contributes to the overall animation output (its weight) and other timing and blending properties.

**Blending**
The action system includes a feature called blending which is primarily used to describe how the individual animation sets should transition between one another when switching between different actions.

Currently, there are only two options for blending: `Off` – which tells the action system that no blending should occur when the animation set is assigned to the controller, and `MixFadeGroup` – which causes a previous set's weight (belonging to the same group) to be decreased, while the new set's weight is increased. This provides a fade-in/fade-out effect between actions.

For many types of actions (although certainly not all) this feature is extremely useful. In many cases, even if we only provide one frame of animation per pose, we can smoothly transition the limbs of a character between those poses without the artist having to provide transition animations. Good candidates are generally animations where the bones are not in wildly different configurations. For example some good blend candidates would typically include Stop to Walk, Stop to Run, Stop to Strafe, Walk to Run, Walk to Strafe, Run to Strafe, Stand to Crouch, (and the reverse for all of these). When the skeletal configurations are very different, blending will not always be a viable solution. For example, going from a swim animation to a walk animation would probably not work all that well and some form of hand crafted transition animation would probably be in order (or just an immediate switch).

While we give up some of the control that hand animated transitions provide, blending usually provides reasonably realistic transitions, with the added advantage of *greatly* reducing animation asset requirements (simple single frame poses will often suffice).

**Groups**

The AAS includes a concept of groups. A group is essentially just a name that we assign to a set definition describing a relationship between animation sets in different actions. This concept grows out of the requirement for the system to understand which animation set you would like to transition *from*, during the blending process.

Groups are defined between actions, rather than being internal to any one action. That is, the grouping is required for steps that are taken when switching to another action. As a result, set definitions belonging to different actions can be grouped, while within any one action only one set definition may belong to a particular group.

For example, imagine that we have two animation sets named 'Walk' and 'Run' which contain keyframes for the legs of the character only. In this example, we may define two separate actions for the act of running and walking separately. When we switch between these two actions (and we would like the legs to blend smoothly between the two) the system needs to know that the 'Walk' animation set is the one that should be blended out. Because these two set definitions would belong to the same group (perhaps with the name 'Legs') it is able to make this determination.

In addition to blending, the group concept is also applied to the various timing properties available via the set definition. These will be explained later.

Now that we know the definitions for the various components of the system, let us have a look at the format of the .act file.

# The ACT File Specification

Below is the ACT file specification. Any dynamic properties are surrounded in braces (*i.e. {Label/Restrictions}* ), while optional values (depending on certain other values used) are shown in ***bold***.

```
 [General]
ActionCount = {0…n}

; ---------------------------------------------------------------------------
; Action Block
; Repeat for each action, the number of which is defined in the
; above 'ActionCount' variable (relative to 1).
; ---------------------------------------------------------------------------
[Action{1…n}]

Name            = {Action Name}
DefinitionCount  = {0…n}


; ---------------------------------------------------------------------------
; Definition Block
; Repeat for each definition, the number of which is defined in the
; above 'DefinitionCount' variable.
; ---------------------------------------------------------------------------
Definition[{0…(DefinitionCount-1)}].SetName      = {Animation Set Name}
Definition[{0…(DefinitionCount-1)}].GroupName  = {Group Name (Unique to this action)}
Definition[{0…(DefinitionCount-1)}].Weight         = {Track Weight}
Definition[{0…(DefinitionCount-1)}].Speed          = {Track Speed}
Definition[{0…(DefinitionCount-1)}].BlendMode   = {Off | MixFadeGroup}
Definition[{0…(DefinitionCount-1)}].TimeMode    = {Begin | MatchGroup | MatchSpecifiedGroup}

Definition[{0…(DefinitionCount-1)}].MixLength          = {Seconds to Blend if MixFadeGroup}
Definition[{0…(DefinitionCount-1)}].TimeGroup        = {Group Name if MatchSpecifiedGroup}
; ---------------------------------------------------------------------------
; End Definition Block
; ---------------------------------------------------------------------------




; ---------------------------------------------------------------------------
; End Action Block
; ---------------------------------------------------------------------------
```

At the top of file is the General block, which describes properties global to the file. In this block there is just a single property we need to specify -- the number of action definitions that are going to follow. Each action definition contains the data for a single action. If the ActionCount property in the General block is assigned a value of 10, this should be followed by 10 Action blocks.

Following the General block are the Action blocks. Each action block should have the name Action*N* where *N* is the number of the block currently being defined. The number of action blocks should be equal to the ActionCount property specified in the General block.

Each action block has two properties that need to be set -- the name of the action and the number of set definitions it will contain. The name assigned to the action is the same name the application will use to apply that action to the actor, so you should use something descriptive (e.g., "Strafe and Shoot"). The DefinitionCount property is where we specify the number of animation sets that will comprise this action. If we set this value to *N*, it should be followed by *N* Definition blocks nested inside the Action block. Each definition contains the name of an animation set and the properties needed to assign it to the animation mixer.

Each Definition block inside an Action block has a name in the format Definition*N*, where *N* is the numerical index of the definition inside the Action block. Unlike the General and Action blocks, each definition block has many properties to set which describe how the animation set is set up.

We will now discuss the properties of the definition block.

```
Definition[{0…(DefinitionCount-1)}].SetName      = {Animation Set Name}
Definition[{0…(DefinitionCount-1)}].GroupName    = {Group Name (Unique to this action)}
Definition[{0…(DefinitionCount-1)}].Weight       = {Track Weight}
Definition[{0…(DefinitionCount-1)}].Speed        = {Track Speed}
Definition[{0…(DefinitionCount-1)}].BlendMode    = {Off | MixFadeGroup}
Definition[{0…(DefinitionCount-1)}].TimeMode     = {Begin | MatchGroup | MatchSpecifiedGroup}
Definition[{0…(DefinitionCount-1)}].MixLength    = {Seconds to Blend if MixFadeGroup}
Definition[{0…(DefinitionCount-1)}].TimeGroup    = {Group Name if MatchSpecifiedGroup}
```

### Set Name

This is the name of the animation set for which this definition is applicable. It should match the name of an animation set in the X file. If you have created the art assets yourself, then you will know the names you have given to your sets, otherwise your artist will supply these names to you. If you are using models that you have downloaded off the internet then you can always convert the X file to a text based X file, open it up in a text editor and search for the Animation Set data objects to retrieve their names.

### Group Name

This is the where you specify the group name you would like to apply to this set definition. There should not be another definition in this action which belongs to the same group. Animation sets in different actions that belong to the same group can be transitioned between when a new action is applied. For example, you might have one definition in every action that animates the legs of the character. You might give each of these definitions in each action the group name of 'Legs'.

### Weight

This is where we specify the weight of the animation set and thus control the strength of its contribution to the actor's hierarchy. You will usually set this to 1.0 so that the animation, when fully transitioned in, will play at the strength originally intended by the asset creator.

### Speed

This is where we specify the speed at which we would like the animation set play when it is first activated. The speed of the track can still be modified by the application after the action has been applied, and we do that very thing in Lab Project 12.3 to speed up the walking animation based on the character's current velocity.

### BlendMode
This property can be set to one of two values; Off or MixFadeGroup.

**Off**
If set to Off, this animation set will not transition to full strength over a period of time and will not be blended with any other animation sets currently playing that have been assigned to the same group. When the action is applied, any sets with the Off BlendMode state will be immediate assigned to a free track at the weight and speed described above. If the previous action was playing an animation set that belonged to the same group, that animation set will be deactivated immediately. There will be no transition from one to the next.

**MixFadeGroup**
When the BlendMode property is set to MixFadeGroup, we are stating that when this action is applied, this animation set's weight should slowly transition to its full strength (specified by the weight property above) over a period of time. At the same time, if the previous action was playing an animation set that has been assigned the same group name, then that animation set will have its weight configured to fade out over this same period of time. During this transition time, both the animation sets will be active and blended together to create transition poses from one animation set to the next. Once the previous animation set has had its weight faded out to zero, it is deactivated and the track it was assigned to is free to be used by the system again.

### Mix Length
This property is only used if the animation set's BlendMode property has been set to MixFadeGroup. It specifies how long we would like this definition to take to transition to full strength. Similarly, it tells us how long any current animation set belonging to the same group should take to transition out.

### TimeMode
The TimeMode property allows us to configure the starting track time for an animation set when it is first applied to the mixer. This is a very useful property that allows us to synchronize the track time of the newly applied animation set with the track time of either a previous animation set from the same group that is transitioning out, or to that of a track playing an animation set belonging to a different group. The latter is also very important as it allows us to synchronize the arms and legs animations even though they belong to different groups. We might imagine that the arms animation set is configured to take its time from the legs animation set, so even when we apply a new animation set to the torso (such as walking), it will take its starting time from the current position of the legs so that the walking movement of both the arms and legs are synchronized.

There are three possible values for this property that our action system understands. They are listed below.

**Begin**
No time matching is performed. When this animation set is applied, the track clock is set such that the animation set starts playing from the beginning. In this mode, the animation set is not synchronized with any other animation sets in the same action and operates in isolation from a timing perspective.

**MatchGroup**
This mode is extremely useful when an animation set is configured to blend. It informs the action system that when this animation set is applied, we do not want to start playing it from the beginning but instead, its periodic position should match the current periodic position of an animation set currently playing (from a previously set action) that belongs to the same group. If we imagine two actions, where one has a 'Walk' animation and another has a 'Run' animation, if both of these were assigned to the legs group, the 'Run' animation could inherit the time currently being used by the 'Walk' set to assure a smooth transition.

**MatchSpecifiedGroup**
This mode is used to ensure synchronization between animation sets within an action that belong to different groups. As an example, we may apply an action to a character that contains an 'Arms' group and a 'Legs' group. The 'Legs' group might be set to MatchGroup mode so that it inherits the time from a current 'Legs' group set to ensure a smooth transition. We might then configure the 'Arms' group in our new action to inherit the time from the 'Legs' group so that the animation sets for both the arms and legs not only remain synchronized with each other, but also inherit their periodic position from a 'Legs' group that was playing in the previous action.

If this mode has been set for a given animation set, the name of the group that you would like this animation set to take its timing information from must also be specified using the TimeGroup property.

### Time Group
This property is needed only for definitions that have their TimeMode property set to MatchSpecifiedGroup. It should contain the name of the group you would like this animation set to inherit its initial track time from.

We have now covered the complete specification of our new .ACT file format and have examined what each property does. This will go a long way towards helping us understand the implementation of our new animation system. While the .ACT file specification does not provide as much flexibility as a scripting system -- where we would be able to include conditional branches based on certain states of the application for instance -- this simple specification does provide us with enough control to be able to achieve a reasonably realistic set of actions and transitioning effects for use in our current demos.

## An .ACT file example

Looking at an example should help to clarify the subject. Below you can see a very simple .act file that contains three actions. Each action contains three set definitions. They describe animation sets that work with the torso, legs and hips. The hips animations were created by the artist to counteract the rotation of the pelvis caused by the leg animations. That is, the hips animation essentially undoes the rotation to the midriff caused by the leg animations. This stops the torso from rotating side to side as the legs walk.

**Note:** Lines that start in an .ini file with a semicolon are ignored by the loading functions. They are comments and equivalent to the C++ double forward slash.

The three actions defined in this .ACT file are ‘Walk_At_Ease’, ‘Walk_Ready’ and ‘Walk_Shooting’. Each action contains three set definitions for the legs, hips and torso. The legs, hips and torso definitions for each action are assigned to the ‘Legs’, ‘Hips’, and ‘Torso’ groups, respectively.

```
; Action Definition Script 'US Ranger.act'
[General]
ActionCount = 3

; ------------------------------------------------------------------------------
; Name : Walk_At_Ease
; Desc : Standard walking with weapon down by side.
; ------------------------------------------------------------------------------
[Action1]

Name                        = Walk_At_Ease
DefinitionCount             = 3

Definition[0].SetName       = Walk_Legs_Only
Definition[0].GroupName     = Legs
Definition[0].Weight        = 1.0
Definition[0].Speed         = 1.0
Definition[0].BlendMode     = MixFadeGroup
Definition[0].MixLength     = 0.3
Definition[0].TimeMode      = MatchGroup

Definition[1].SetName       = Walk_Variation
Definition[1].GroupName     = Hips
Definition[1].Weight        = 1.0
Definition[1].Speed         = 1.0
Definition[1].BlendMode     = Off
Definition[1].TimeMode      = MatchSpecifiedGroup
Definition[1].TimeGroup     = Legs

Definition[2].SetName       = Gun_At_Ease
Definition[2].GroupName     = Torso
Definition[2].Weight        = 1.0
Definition[2].Speed         = 1.0
Definition[2].BlendMode     = MixFadeGroup
Definition[2].MixLength     = 1.0
Definition[2].TimeMode      = MatchSpecifiedGroup
Definition[2].TimeGroup     = Legs

; ------------------------------------------------------------------------------
; Name : Walk_Ready
; Desc : Standard walking with weapon at the ready (clasped in both hands)
; ------------------------------------------------------------------------------
[Action2]

Name                        = Walk_Ready
DefinitionCount             = 3

Definition[0].SetName       = Walk_Legs_Only
Definition[0].GroupName     = Legs
Definition[0].Weight        = 1.0
Definition[0].Speed         = 1.0
```

```
Definition[0].BlendMode     = MixFadeGroup
Definition[0].MixLength     = 0.3
Definition[0].TimeMode      = MatchGroup

Definition[1].SetName       = Walk_Variation
Definition[1].GroupName     = Hips
Definition[1].Weight        = 1.0
Definition[1].Speed         = 1.0
Definition[1].BlendMode     = Off
Definition[1].TimeMode      = MatchSpecifiedGroup
Definition[1].TimeGroup     = Legs

Definition[2].SetName       = Gun_Ready_Pose
Definition[2].GroupName     = Torso
Definition[2].Weight        = 1.0
Definition[2].Speed         = 1.0
Definition[2].BlendMode     = MixFadeGroup
Definition[2].MixLength     = 1.0
Definition[2].TimeMode      = Begin



; ------------------------------------------------------------------------------
; Name : Walk_Shooting
; Desc : Standard walking with weapon raised in a shooting pose.
; ------------------------------------------------------------------------------
[Action3]

Name                        = Walk_Shooting
DefinitionCount             = 3

Definition[0].SetName       = Walk_Legs_Only
Definition[0].GroupName     = Legs
Definition[0].Weight        = 1.0
Definition[0].Speed         = 1.0
Definition[0].BlendMode     = MixFadeGroup
Definition[0].MixLength     = 0.3
Definition[0].TimeMode      = MatchGroup

Definition[1].SetName       = Walk_Variation
Definition[1].GroupName     = Hips
Definition[1].Weight        = 1.0
Definition[1].Speed         = 1.0
Definition[1].BlendMode     = Off
Definition[1].TimeMode      = MatchSpecifiedGroup
Definition[1].TimeGroup     = Legs

Definition[2].SetName       = Gun_Shooting_Pose
Definition[2].GroupName     = Torso
Definition[2].Weight        = 1.0
Definition[2].Speed         = 1.0
Definition[2].BlendMode     = MixFadeGroup
Definition[2].MixLength     = 1.0
Definition[2].TimeMode      = Begin
```

In each action, the same animation set for the legs group is being used. This is the animation set with the name 'Walk_Legs_Only'. As an action is applied, the track position for the 'Walk_Legs_Only' animation set in that action will take its timing from a previous set assigned to the same group that is currently playing. In this simple example, where each action uses the same animation set for the legs, it simply means that as the 'Walk_Legs_Only' animation set is set on the mixer for a given action, its initial position will be set to the 'Walk_Legs_Only' set that was being played by the previous action. The 'Legs' group in each action is also set to MixFadeGroup so that the animation set will fade in over the period specified in the MixLength parameters (0.3 seconds in this example). In this particular .ACT file, all actions use the same animation set for the legs, so the transition is a mostly futile. However, we might want to add an additional action such as 'Idle_At_Ease' which would use a different animation set for the legs; perhaps one in which the character was no longer walking. In such a case, the MixFadeGroup property would ensure that when we transition from the 'Idle_At_Ease' action to the 'Walk_At_Ease' action, there would be a 0.3 second transition while the legs move from the idle pose into the walking pose. However, if we know in advance that we only wanted to use walking poses (unlikely) then we could set the BlendMode to 'Off'.

The 'Hips' group animation set in each action does not use blending. Once again, it is the same set being used by the 'Hips' group in each action. As we discussed in the previous paragraph, in this section we are assuming that there is only one 'Hips' animation set used by all actions, so there is no need to blend between 'Hips' animation sets in different actions. In each action however, the 'Hips' group animation set is configured to take its timing from the 'Legs' group. This is very important in this instance because the 'Hips' animation set is really nothing more than an inverse transformation applied to the pelvis to undo any rotation applied to it by the walking of the legs. That is, the artist had to rotate the pelvis bone to make the legs animate properly, but this inverse transformation is applied to the next bone up to stop the pelvis rotation (which is the parent bone) being reflected up to the torso.

Finally, the last animation set specified in each action is the one that belongs to the 'Torso' group. A different animation set is used in each action for this purpose. In the first action ('Walk_At_Ease'), the 'Torso' animation set is called 'Gun_At_Ease' and it animates the torso such that the character is walking with his gun in his hand, with his arms swinging back and forth in time with his legs. Because of this, in the first action, the timing of this animation set is taken from the 'Legs' group so that the swinging arms are synchronized with the walking legs.

In the second action, the animation set in the 'Torso' group is called 'Gun_Ready_Pose'. This adjusts the torso of the character so that he is holding the gun to his chest and looking alert, but not yet assuming a hostile posture. Because this animation no longer needs to be synchronized with the movements of the legs (the arms are no longer swinging back and forth), the timing mode of this animation is set to 'Begin'. It will begin playing from the beginning as soon as this action is activated. Notice however that it is still set to blend with other sets in its group. This means, if the previous action playing was 'Walk_At_Ease', when this new action is applied, the arms would slowly transition from swinging back and forth up to the ready position over a period of one second.

The third action has a 'Torso' group animation set called 'Gun_Shooting_Pose'. This is a single frame animation of the torso holding the gun up to his face in an aiming posture. Since this pose requires no relationship with the animations being played out on other parts of the character's body, its time mode is set to 'Begin'. It is also set to blend in from animation sets within the same group. That is, if this action

is applied and the previous action being played was 'Walk_Ready', the torso of the character would transition from the ready pose to the shooting pose over a period of one second.

We have now covered all the properties that our Action Animation System will have to work with. We have also seen the format in which the data will be presented to our application. We now know how to write .ACT files in a text editor and the format in which the contents need to be laid out. So let us now begin coding the Action Animation System.

# The Action Animation System – Implementation

The structures we will add to CActor will very much mirror the layout of the *.ini* file (.ACT file). The actor will maintain an array of CActionDefinition structures and each action definition structure will contain the information for a single action loaded from the .ACT file. A LoadActionDefinitions function will be added to the CActor interface. This method, which will be called from the application and passed the name of an .ACT file, will load the data and store it internally. For each action definition in the .ACT file, a CActionDefinition structure will be allocated, populated, and added to the actor's CActionDefinition array.

Each CActionDefinition structure will maintain an array of SetDefinition structures. Each set definition will contain the data for a single animation set when the action is applied. As you can see, this follows the exact layout of the .ACT file. The .ACT contains a number of action definitions and each action definition contains a number of set definitions. In code, the actor contains an array of action definitions and each action definition contains an array of set definitions. The ApplyAction method will also be added to the interface of CActor so that the application can inform the actor to apply one of its actions at any time.

Of course, simply maintaining an array of the information loaded from the .ACT file is not enough by itself to describe the state that the controller may be in at any given time to our action system. For example, we will need to know, for each track on the mixer, is there currently an animation set assigned to it or is it free for our system to use? Is the track in the phase of transitioning out? What was the name of the animation set assigned to that track and is it used by the current action we are applying? A support structure of some kind needs to be introduced. We will need this anyway for us to quickly ascertain during the blending process, which group the animation set assigned to a given track belongs to. We know for example that if the time mode of the animation set we are about to assign has been set to MatchGroup, we will need to search the tracks of the animation mixer for an animation set that belongs to the same group. As the mixer has no concept of groups, we will obviously need a support structure that will store this information for each track.

The support class we will use is called CActionStatus. This will be a new member of CActor which contains the status of each track on the animation mixer. Internally, the CActionStatus class will manage an array of TrackActionStatus structures; one for each track on the mixer. Each TrackActionStatus structure will contain the current properties for a given track on the animation mixer.

To clarify, if the actor had a 15 track animation mixer, the actor would contain a single CActionStatus member that would contain an array of 15 TrackActionStatus structures. Each TrackActionStatus structure contains information about whether its associated track is currently in use by the action system, the name of the set currently assigned to it, the group that set belongs to, and so on.

The following diagram shows the relationships between CActor and the various structures just discussed.

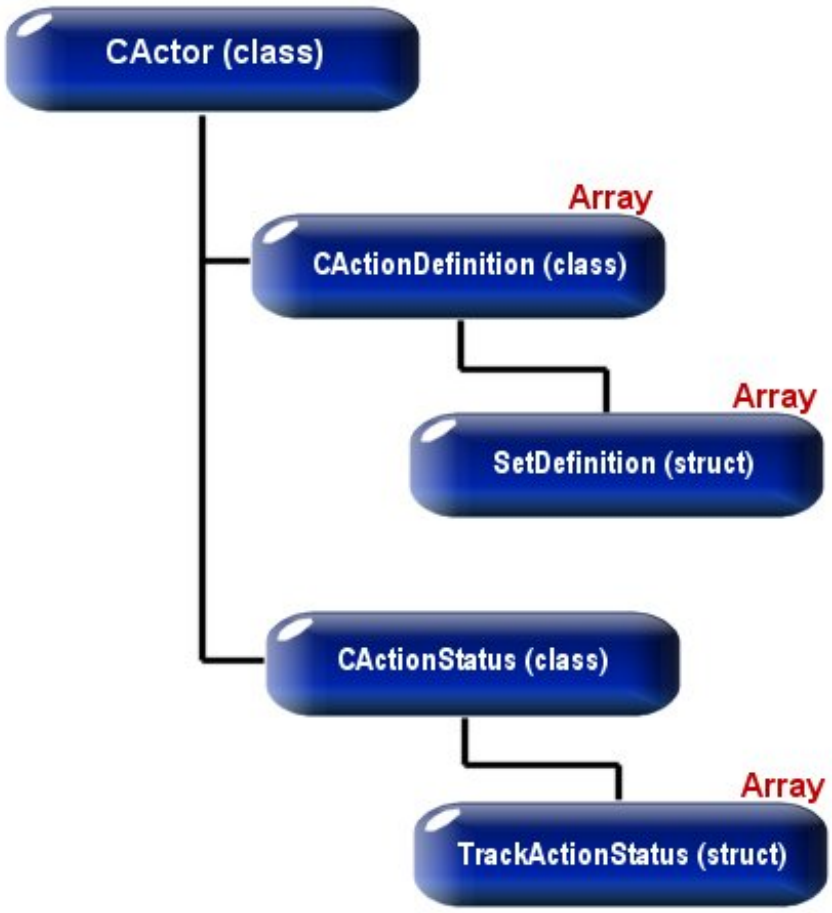


**Figure 12.5**

Notice that there is a single CActionStatus member that contains the status of the current controller being used by the actor. As the controller is essentially swapped out when referencing is being used, we will also have to do the same with the CActionStatus object. That is, each CObject reference to an actor will now contain not only its own animation controller but also its own CActionStatus object describing the status of all the tracks on its controller's mixer. Our actor's AttachController method will now accept both a pointer to the controller we wish to attach and the CActionStatus object that describes the status of that controller's tracks. This will be a very small adjustment which we will discuss later.

The CActionDefinition array will be allocated when the user calls the CActor::LoadActionDefinitions function. There will be an element in this array for each action defined in the file. Let us now have a look at the changes to CActor.

## The CActor Class – Code Changes

We have added some new member variables to CActor to store the data of the action system. As our CActor class definition is now getting quite large, we have only shown the new or modified variables and functions that we are going to introduce in this workbook.

***Excerpt from CActor.h***

```
class CActor
{
…
…
 private:

    // New 'Action' set data variables
    CActionDefinition   *m_pActions;        // An array of loaded actions
    ULONG                m_nActionCount;    // The number of actions
    CActionStatus       *m_pActionStatus;   // Stores the status of all tracks
…                                           // for the action system
…
  public :
    // New 'Action' Methods
        HRESULT           LoadActionDefinitions       ( LPCTSTR FileName );
        HRESULT           ApplyAction                 ( LPCTSTR ActionName );
        CActionDefinition * GetCurrentAction          ( ) const;
        CActionStatus     * GetActionStatus           ( ) const;
…
…
    // Existing functions modified by action system
    LPD3DXANIMATIONCONTROLLER DetachController( CActionStatus ** ppActionStatus );

    void AttachController    ( LPD3DXANIMATIONCONTROLLER pController,
                               bool bSyncOutputs = true,
                               CActionStatus * pActionStatus = NULL );

    HRESULT SetActorLimits      (       ULONG MaxAnimSets = 0,
                                        ULONG MaxTracks = 0,
                                        ULONG MaxAnimOutputs = 0,
                                        ULONG MaxEvents = 0,
                                        ULONG MaxCallbackKeys = 0 );
  }
```

Let us first discuss the three new member variables that have been added.

**CActionDefinition *m_pActions**
This is a pointer to an array of CActionDefinition objects that will be allocated when the application issues a call to the CActor::LoadActionDefinitions method. Each element in this array is a class that manages the data for a single action loaded from the .ACT file. The CActionDefinition class exposes methods of its own, such as LoadAction and ApplyAction, which the actor uses to perform the heavy lifting associated with loading or applying an action. The actor methods are really just lightweight wrappers that make calls to the methods of the current action being loaded or applied. We will look at the members and methods of the CActionDefinition class shortly.

**ULONG m_nActionCount**
This member is set in the CActor::LoadActionDefinitions method and contains the number of actions defined in the .ACT file. Therefore, it also describes the size of the above array.

**CActionStatus *m_pActionStatus**
This is a pointer to a single CActionStatus object that will contain the current states of each track on the animation mixer. It is used by the Action System to remember which animation sets have been assigned to which tracks, which groups those animation sets belong to, and the status of those tracks (e.g., if they are in the middle of a transition).

We will now discuss the new methods that have been added to CActor. Since most of these functions are light wrappers around calls to the methods of the CActionStatus and the CActionDefinition objects, we will not get the full picture until these objects have been covered as well. However, it should be fairly obvious from a high level, with a little explanation, what these methods are doing.

## CActor::LoadActionDefinitions

This method is called by the application after the actor has been allocated and populated by the mesh and animation data from the X file. This function is called and passed the name of an .ACT file that contains the data for the action system. This function will open that file, allocate a CActionDefinition array large enough to contain the correct number of actions, and then populate this array with the action data from the file. It will also allocate a new CActionStatus object that has a large enough internal TrackActionStatus array to store the status of every track on the animation mixer.

The function is passed the name of the file we wish to load the action definitions from. The first thing the function does is delete any previous CActionDefinitions array that may exist in the actor prior to this call. It also releases its CActionStatus pointer if one already exists. Notice that because our CActionStatus object will have to be attached and detached from the controller and stored in an external object when referencing is being used, we have implemented the COM-style AddRef and Release mechanism for this object.

```
HRESULT CActor::LoadActionDefinitions( LPCTSTR FileName )
{
    ULONG   ActionCount = 0, i;
    HRESULT hRet;

    // Clear any previous action definitions
    if ( m_pActions ) delete []m_pActions;
    m_pActions     = NULL;
    m_nActionCount = 0;

    // Clear any previous action status
    if ( m_pActionStatus ) m_pActionStatus->Release();
    m_pActionStatus = NULL;
```

We will now retrieve the ActionCount property from the .ACT file. Recall from our studies in Module I that we use the Win32 function GetPrivateProfileInt to read in integer data from an .ini text file.

```
    // Retrieve the number of actions stored in the file
    ActionCount = ::GetPrivateProfileInt( _T("General"),
                                          _T("ActionCount"),
                                          0,
                                          FileName );

    if ( ActionCount == 0 ) return D3DERR_NOTFOUND;
```

Just to recap, the first parameter is the name of the data block in the file that we wish to retrieve the value from. Recall from our .ACT format that the action count property is set in the General block. As the second parameter we pass in the name of the property we wish to fetch the value for (ActionCount). If this property is not found, then the third parameter specifies a default value to be used. In this instance, if the ActionCount property is missing we have an invalid .ACT file and return a default action count of 0 which will cause the function to return without loading any action data. As the final parameter, we pass in the name (and path) of the file we wish to extract the value from.

Now that we know how many actions there are, we can allocate the actor's CActionDefinition array to the correct size so that there is an element for each action in the file. We also allocate the CActionStatus object.

```
    // Allocate the new actions array
    m_pActions = new CActionDefinition[ ActionCount ];
    if ( !m_pActions ) return E_OUTOFMEMORY;

    // Allocate the new status structure
    m_pActionStatus = new CActionStatus;
    if ( !m_pActionStatus ) return E_OUTOFMEMORY;
```

The CActionStatus object will maintain an array of TrackActionStatus objects for each track on the mixer. Each one will contain the status of a track. Let us now fetch the number of tracks available on this actor's animation mixer and send this value to the CActionStatus::SetMaxTrackCount method. We will cover this method in a moment, but it essentially allocates the array of TrackActionStatus structures to be as large as described by the input parameter. We will also copy over the action count property that we extracted from the .ACT file.

```
    // Setup max tracks for action status
    m_pActionStatus->SetMaxTrackCount( (USHORT)GetMaxNumTracks() );

    // Store the count
    m_nActionCount = ActionCount;
```

It is now time to load each action in the .ACT file into each element in the CActionDefinitions array we just allocated. To do this, we loop through each action definition in the array and call its LoadAction method. As you can see, it is the CActionDefinition::LoadAction function which actually loads the data from each action in the .ACT file. We pass it two parameters. The first is a numerical index that, when appended to the word 'Action', will describe the name of the Action block in .ACT file that the current action needs to fetch its data from. As the second parameter, we simply pass the file name of the .ACT file that contains the action data.

```
    // Process the actions
    for ( i = 0; i < ActionCount; ++i )
    {
        // Load the definition from file
        hRet = m_pActions[i].LoadAction( i + 1, FileName );
        if ( FAILED(hRet) ) return hRet;

    } // Next Action in file

    // Success!
    return S_OK;
}
```

And that is it for this function. Obviously, many of the blanks will be filled in when we cover the code to the CActionDefinition methods and the CActionStatus methods. However, by examining the actor's interaction with these objects first, we will have a better understanding of their place within the entire system when we do get around to covering them.

## CActor::GetCurrentAction

The next CActor method we will cover is a useful function should the application wish to query which action is currently being played by the actor. Examining the code also establishes the responsibility of the actor's CActionStatus member to keep a record of the current action that has been applied.

```
CActionDefinition * CActor::GetCurrentAction( ) const
{
    // Return the current action if status is available
    if ( !m_pActionStatus ) return NULL;
    return m_pActionStatus->GetCurrentAction();
}
```

Notice that it is the CActionStatus structure that stores the current action (the action that was last applied) and this property can be fetched via CActionStatus::GetCurrentAction. The CActionStatus object will be covered in a moment, but is really little more than a storage container with functions to set and retrieve its data.

## CActor::GetActionStatus

This function allows the application to fetch a pointer to the CActionStatus object being used by the actor. This is useful since the CActionStatus object contains the status information for each track of the animation controller. If the application wanted to play an additional animation that was not part of the action, it could fetch the CActionStatus object, use it to find a free track, and flag that track as being in use. If we did not do this, then the action system would have no way of knowing that one of the tracks on the animation mixer was currently being used by an animation set external to the action system and erroneously think it could use that track the next time an action was applied.

```
CActionStatus * CActor::GetActionStatus( ) const
{
    // Validate requirements
    if ( !m_pActionStatus ) return NULL;

    // Add a reference to the action status, we have a floating pointer
    m_pActionStatus->AddRef();

    // Return the pointer
    return m_pActionStatus;
}
```

Another reason to fetch the CActionStatus object is to store it in a CObject. That is, when multiple CObjects reference the same CActor, each object will contain its own controller and status object that will be set before that object is animated.

## CActor::ApplyAction

This next function is called by the application when it wishes to change the current action being played to a new one. All the application has to do is call this function and specify the name of the action it wishes to apply. This will be the name assigned to one of the actions in the .ACT file.

This function is the primary runtime interface to the Action Animation System. Once the action data has been loaded, it is the only function that needs to be called to apply new actions to the controller. This function is rather small, since most of the core work takes place inside the CActionDefinition::ApplyAction method (just as most of the loading of data was performed inside this object) which we will examine later.

The first thing this function does is test to see that the actor has an animation controller and a CActionStatus object defined. If not, it returns, since there is no controller to configure with action data. We then get a pointer to the current action being played and test to see if its name is the same as the name passed in. If it is, then the application has asked for an action to be played which is already playing, and there is nothing to do. In this case we will simply return.

```
HRESULT CActor::ApplyAction( LPCTSTR ActionName )
{
    ULONG   i;
    HRESULT hRet;

    // Bail if there is no active controller
    if ( !m_pAnimController || !m_pActionStatus ) return D3DERR_INVALIDCALL;

    // If this action is already set, ignore
    CActionDefinition * pCurrentAction = m_pActionStatus->GetCurrentAction();

    if ( pCurrentAction && _tcsicmp( pCurrentAction->GetName(), ActionName )== 0 )
        return D3D_OK;
```

Notice that it is the CActionStatus object which contains a pointer to the current CActionDefinition that has been applied (we can get a pointer to it via its GetCurrentAction method). Once we have a pointer to the current CActionDefinition, we can use its GetName function to retrieve a string containing its name. In the above code, we use the _tcsicmp string comparison function to compare the name of the current action with the name of the action the application would like to apply. If the _tcsicmp function returns zero, the two strings are identical and there is no need to do anything.

In the next section of code we have determined that the application would like to set a new action. The first thing we must do is loop through each action in the actor's CActionDefinition array and search for the one with a name that matches the action name passed into the function. Here is the remainder of the function:

```
    // Loop through each action
    for ( i = 0; i < m_nActionCount; ++i )
    {
        CActionDefinition * pAction = &m_pActions[i];

        // Compare the name, and apply if it matches
        if ( _tcsicmp( ActionName, pAction->GetName() ) == 0 )
        {
            // Apply the action
            hRet = pAction->ApplyAction( this );
            if ( FAILED(hRet) ) return hRet;

            // Store the action
            m_pActionStatus->SetCurrentAction( pAction );

            // Success!
            return D3D_OK;

        }  // End if names match

    } // Next action

    // Unable to find this action
    return D3DERR_NOTFOUND;
}
```

Once we find a matching action in the CActionDefinition array we will use its ApplyAction method to apply the action to the controller (covered in a moment). This is obviously where the core work of applying the animation sets to the controller and setting blending sequences will take place. After we have applied the current action, we must let the CActionStatus object know that there is a new current action. We do this using the CActionStatus::SetCurrentAction method. This method just stores the passed pointer in a member variable. As discussed, the CActionStatus object is just storage object that has its properties set and retrieved via a few member functions.

Before we get to the core code inside the CActionDefinition object, we will first discuss the layout and methods of the CActionStatus object since it is used extensively by the CActionDefinition::ApplyAction function.

## The CActionStatus Class

The CActionStatus class stores the properties of each track on the animation mixer. For a given controller only one CActionStatus object will exist. Internally, the object has only four member variables. It stores the number of tracks on the controller, the current action that is being used by the controller, its own reference count (it uses COM-style lifetime encapsulation semantics), and a pointer to an array of TrackActionStatus structures. There will be one of these structures for each available track on the animation mixer and each structure stores five bits of information about a given track.

Below is the class definition for CActionStatus. Notice that the TrackActionStatus structure is defined in the CActionStatus namespace. Take a look at the member variables and functions that are specified as part of this namespace and also notice that some of the simpler methods are inline.

***Excerpt from CActor.h***

```
class CActionStatus
{
public:

    struct TrackActionStatus // Status of a single track
    {
        bool   bInUse;              // This track is in use
        bool   bTransitioningOut;   // The track is currently transitioning out
        TCHAR  strGroupName[127];   // The name of the group assigned to this track
        TCHAR  strSetName[127];     // The name of the set assigned to this track

        const CActionDefinition * pActionDefinition;  // The action definition
                                                      // to which this track
                                                      // applies

         ULONG                    nSetDefinition;     // The set definition index,
                                                      // of the above action, to
                                                      // which this track applies
    };

    // Constructors & Destructors for This Class.

    CActionStatus( );
    virtual ~CActionStatus( );

    // Public Functions for This Class.

    ULONG               AddRef           ( );
    ULONG               Release          ( );
    HRESULT             SetMaxTrackCount ( USHORT Count );
    void                SetCurrentAction ( CActionDefinition * pActionDefinition);
    USHORT              GetMaxTrackCount ( ) const  { return m_nMaxTrackCount; }
    TrackActionStatus * GetTrackStatus   ( ULONG Index ) const
                                          {
                                            return ( Index < m_nMaxTrackCount ) ?
                                            &m_pTrackActionStatus[ Index ] :
                                            NULL;
                                           }
```

```
    CActionDefinition * GetCurrentAction ( ) const   { return m_pCurrentAction; }

private:

    // Private Variables for This Class.

    USHORT              m_nMaxTrackCount;     // The maximum tracks for controller
    TrackActionStatus * m_pTrackActionStatus; // Array of track status structures
    CActionDefinition * m_pCurrentAction;     // The current action applied
    ULONG               m_nRefCount;          // Reference count variable
};
```

This class has only four member variables, so let us discuss them before we look at the functions.

**USHORT m_nMaxTrackCount**
This member will contain the number of tracks on the animation mixer. This value is set when the actor calls the CActionStatus::SetMaxTrackCount method when the actor is first loaded, or when the application resizes the limits of the actor's controller using CActor::SetActorLimits.

Any time the actor clones its controller and extends the number of tracks, the CActionStatus::SetMaxTrackCount method will be called so that the m_nMaxTrackCount member can have its value updated and the TrackActionStatus array (discussed below) can be resized to contain this many elements.

**TrackActionStatus * m_pTrackActionStatus**
This member is a pointer to an array of TrackActionStatus structures. There will be one TrackActionStatus structure in this array for every track on the animation mixer. That is, the size of this array will be equal to the value of m_nMaxTrackCount discussed above.

Just like the previous value, this array has its size set in the CActionStatus::SetMaxTrackCount method, which is called by the actor whenever the controller is first created or its maximum track limits are extended (via cloning).

Each TrackActionStatus structure in the array contains five very important pieces of information about each track that our Action System will need to know in order to apply actions successfully. Let us discuss the members of this structure.

> **bool bInUse**
> This is a simple boolean member which specifies whether the associated mixer track is currently in use. When an action is applied, the animation sets referenced by that action will need to be assigned to tracks on the animation mixer. The action system will need to know which tracks are currently free and can have those sets assigned to them.
>
> If this member is set to false then the track is free to have an animation set assigned to it. If it is set to true then it means it is being used or was being used by a previous action and is still in the process of having its weight faded out.

**bool bTransitioningOut**

This boolean is set to true if the associated track is currently in the process of being faded out. When an action is applied, and one of the sets in that action shares a group name with a previously assigned animation set, the previous animation set will be set to transition out over a period of time if blending is enabled for the new set. The previous set of the same group name will have a sequencer event activated for it that will fade the track's weight to zero over a specified period of time. This boolean will then be set to true so that the action system knows that this animation set and its track must be left alone to fade out. A sequencer event will also be set for the track at the end of the fade out period that disables the track on the mixer. The next time a new action is applied, we can search the tracks that have this boolean set to true and if we discover that its associated track has been disabled, we can once again mark this track as no longer in use. This is done by setting the bInUse Boolean and the bTransitioningOut Boolean to false.

The need for this boolean will become clear when we cover the code to the CActionDefinition::ApplyAction function. Essentially, it is our way of saying that this animation set was at some time in the past set to transition out over a period of time so leave it alone until that is done, regardless of what other actions may be applied in the fade out period and what their blend settings may be. Once an animation set has been configured to fade, it essentially gets left to its own devices by the action system until the fade out is complete. Then the track can once again be used by future actions.

**TCHAR strGroupName[127]**

As discussed earlier, one of the key properties for the blending process between actions is the group name assigned to an animation set used by those actions. If an action is applied which contains an animation set that shares the group name with an animation set in the previously active action, and the animation set in the new action has been configured to blend, the action system will fade the previous animation set out while simultaneously fading the weight of the new animation set in.

The action system performs this task by recognizing that for any action there should only be a single track from the same group. Before an animation set is assigned, we first search the tracks of the mixer to see if a track existed in the previous action that shares the same group. If the new action is configured to group blend, the previous track will be faded and the new track introduced over the same period of time. Obviously, for the action system to perform this blending it must be able to search the tracks of the controller and see if tracks exist with the same group name. It is in this member that we store a string containing the group name of the set definition assigned to this track. When the CActionDefinition::ApplyAction method assigns the animation sets of an action to the tracks on the mixer, it will also store the group name associated with that set definition within the action in this member.

**TCHAR strSetName[127]**

When the action system assigns an animation set to a mixer track it will store the name of the animation set in this member. Although this information could be fetched by querying the animation set interface assigned to the track, it is more convenient to store it here. If the application wishes to use the CActor::GetActionStatus method to get a pointer to the

CActionStatus object, it will be able to easily examine the TrackActionStatus structures for each track to determine which animation sets are currently being used by the controller and the specified action.

**const CActionDefinition * pActionDefinition**
This member is set when the animation set is assigned to the mixer track. It contains a pointer to the action that set this track. This is useful to have because we know that due to the blending process, not all the animation sets currently set on the mixer will have been set by the currently applied action. If we imagine a case where we have three actions that each use three different animation sets which are configured to blend over a period of 20 seconds, and we apply these three actions over a period of 5 seconds, at 10 seconds (for example) there will actually be 9 animation sets currently in use. The first three will have been set by the first action but will still be in the process of fading out. The second three animation sets will have been set by the action that was applied second and these too will still be in the process of fading out. The final three sets will have been set by the currently applied action. If each track stores a pointer to the action that was responsible for setting its animation set, this may come in handy if the application needs to know this information.

**ULONG nSetDefinition**
This stores the index of the set definition. For example, if the animation set was referenced by the second set definition in the action (inside the .ACT file) this would be the index that was stored here. Once again, this is really for the purpose of providing as much information as we can to the application in the event that it needs it for some reason.

**CActionDefinition *m_pCurrentAction**
This member contains a pointer to the currently set action. This is set and retrieved by the CActionStatus::SetCurrentAction and CActionStatus::GetCurrentAction.

We saw the SetCurrentAction method of this object being called from the CActor::ApplyAction method discussed recently. The function first applied the action and then informed the CActionStatus object of the new action that was set using this method. We also saw how the CActor::GetCurrentAction method simply wraps the call to CActionStatus::GetCurrentAction. These Get and Set functions simply set and return the value of this pointer.

**ULONG m_nRefCount**
Contains the current reference count of the object.

Let us now discuss the methods of the CActionStatus structure. For the most part these will be simple member access functions. We will not cover the AddRef and Release methods here since at this point in the series you should be comfortable with what these methods look like.

## CActionStatus:: CActionStatus

The constructor initializes all member variables to NULL or zero, except for the reference count of the object. Since we are in the constructor, the object is being created, so it must have a reference count of 1 when this function returns.

```
CActionStatus::CActionStatus( )
{
    // Clear required variables
    m_pCurrentAction     = NULL;
    m_pTrackActionStatus = NULL;
    m_nMaxTrackCount     = 0;

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

## CActionStatus::~CActionStatus

The destructor releases the TrackActionStatus array (if it exists) and sets the three other members to NULL or zero.

```
CActionStatus::~CActionStatus( )
{
    // Delete any flat arrays
    if ( m_pTrackActionStatus ) delete []m_pTrackActionStatus;

    // Clear required variables.
    m_pCurrentAction     = NULL;
    m_pTrackActionStatus = NULL;
    m_nMaxTrackCount     = 0;
}
```

## CActionStatus::SetMaxTrackCount

This function resizes the TrackActionStatus array to the size specified by the Count parameter. It is called by the actor whenever the limits of the controller change. There should always be a 1:1 mapping between tracks on the mixer and TrackActionStatus structures in the internal array.

The function first tests the value of the Count parameter. If it is set to zero then it means the caller wishes to flush the per track data currently stored in the CActionStatus object. When this is the case, we delete the TrackActionStatus array and set the m_nMaxTrackCount member variable to zero.

```
HRESULT CActionStatus::SetMaxTrackCount( USHORT Count )
{
```

```
    TrackActionStatus * pBuffer = NULL;

    // If we're clearing out.
    if ( Count == 0 )
    {
        // Just release and return
        if ( m_pTrackActionStatus ) delete []m_pTrackActionStatus;
        m_pTrackActionStatus = NULL;
        m_nMaxTrackCount     = 0;

        // Success
        return S_OK;

    } // End if count == 0
```

In the next section of code, we allocate a new array of TrackActionStatus structures. Its size will equal the size requested by the Count parameter. We then initialize this array to zero.

```
    // Allocate enough room for the specified tracks
    pBuffer = new TrackActionStatus[ Count ];
    if ( !Count ) return E_OUTOFMEMORY;

    // Clear the buffer
    memset( pBuffer, 0, Count * sizeof(TrackActionStatus) );
```

If this CActionStatus object already has data in its TrackActionStatus array then this must be copied into the new array. This allows us to change the number of tracks the CActionStatus object can handle without losing any data that might already exist. This is obviously very important if the resize is performed while the action system is in use. After all, the application can call the CActor::SetActorLimits method at any time (which will cause this function to be called and passed the new track limit of the controller).

As you can see in the following code, if the current track count of the object is larger than zero and its TrackActionStatus member is not NULL then there must be data already and we should copy it over.

```
    // Was there any previous data?
    if ( m_nMaxTrackCount > 0 && m_pTrackActionStatus )
    {
        // Copy over as much data as we can
        memcpy( pBuffer,
                m_pTrackActionStatus,
                min(m_nMaxTrackCount, Count) * sizeof(TrackActionStatus) );

    } // End if existing data
```

Now that we have our new array and have the previous track data copied into it, we can delete our original array.

```
    // Release previous buffer
    if ( m_pTrackActionStatus ) delete []m_pTrackActionStatus;
```

Now we can assign the m_pActionStatus member to point at our new array instead. We also set the m_nMaxTrackCount member to the new track count of the controller that was passed in. The value in m_nMaxTrackCount should always reflect the number of elements in the m_pActionStatus array.

```
    // Store buffer, and count
    m_pTrackActionStatus = pBuffer;
    m_nMaxTrackCount     = Count;

    // Success!
    return S_OK;
}
```

## CActionStatus::SetCurrentAction

This simple function assigns the passed CActionDefinition pointer as the current action. We saw this function being called earlier from CActor::ApplyAction. It first called the CActionDefinition::ApplyAction method to set up the controller for the action being applied and then it called this function to notify the CActionStatus object that the current action has changed.

There is also a 'get' version of this function that is inline; it returns the current action.

```
void CActionStatus::SetCurrentAction( CActionDefinition * pActionDefinition )
{
    // Just store the current action
    m_pCurrentAction = pActionDefinition;
}
```

So there is nothing very complicated about the CActionStatus object. It basically just provides the actor with a means to record the current state of the action system. It stores the current action that has been applied and information about the status of each track on the mixer.

As mentioned earlier, all of the hard work is done inside the CActionDefinition object. This class has two main functions that are called by the CActor. The CActionDefinition::LoadAction method was called from CActor::LoadActionDefinitions once for each action. It has the task of loading the information for the specified action from the .ACT file. The second important function is the CActionDefinition::ApplyAction function which is called by CActor::ApplyAction to actually perform the task of setting up the controller's tracks for the new action. This function is effectively the entire runtime action system. It sets up the transitions between sets that need to be faded in and out, it sets sequencer events to control those fade outs, and a host of other things. Let us now discuss the final component of our action system.

# The CActionDefinition Class

This new class is designed for the storage and application of a single action. When the ACT file is loaded, each individual action (Walk, Run, etc.) is stored within a separate instance of a CActionDefinition object, inside the CActor member array. In addition, the member functions belonging to this class are used directly whenever an action is to be applied to the animation controller.

The class declaration is shown below and is contained in CActor.h. It may seem on first glance that this class is rather large and complex but do not be misled by the two enumerations and one new structure (used by the class) defined in its namespace. Actually, the class has only four member variables and five methods (not including the constructor and destructor) and all but two of those are simple access functions.

```
class CActionDefinition
{
public:

    // Public Enumerators for This Class.
    enum ActBlendMode                       // Flags for SetDefinition::BlendMode
    {
        Off          = 0,
        MixFadeGroup = 1
    };

    enum ActTimeMode                        // Flags for SetDefinition::TimeMode
    {
        Begin               = 0,
        MatchGroup          = 1,
        MatchSpecifiedGroup = 2
    };

    // Public Structures for This Class.

    struct SetDefinition
    {
        TCHAR        strSetName[128];       // The name of the animation set
        TCHAR        strGroupName[128];     // The 'group' to which this belongs.
        float        fWeight;               // Anim Set strength
        float        fSpeed;                // The speed of the set itself
        ActBlendMode BlendMode;             // Blending to use when set is applied
        ActTimeMode  TimeMode;              // Timing Source when set is applied
        float        fMixLength;            // Length of transition is blend
                                            // enabled
        TCHAR        strTimeGroup[128];     // If the time mode specified that we
                                            // should match the timing to a
                                            // specific alternate group, that
                                            // group name is stored here.

        // Internal Temporary Vars
        bool         bNewTrack;             // Assign this set to a new track
        double       fNewTime;              // Use this time for the new track.
    };
```

```
    // Constructors & Destructors for This Class.

    CActionDefinition( );
    virtual ~CActionDefinition( );


    // Public Functions for This Class.

    HRESULT         LoadAction              ( ULONG nActionIndex,
                                              LPCTSTR ActFileName );

    HRESULT         ApplyAction             ( CActor * pActor );

    // Accessor functions
    LPCTSTR     GetName                 ( ) const  { return m_strName; }
    ULONG       GetSetDefinitionCount   ( ) const{ return m_nSetDefinitionCount; }

    SetDefinition * GetSetDefinition( ULONG Index ) const
                                    { return ( Index < m_nSetDefinitionCount )?
                                      &m_pSetDefinitions[ Index ] :
                                      NULL; }


private:

    // Private Variables for This Class.

    TCHAR           m_strName[128];         // The name of this action definition
    USHORT          m_nSetDefinitionCount;  // The number of set definitions
    SetDefinition * m_pSetDefinitions;      // Array of set definition structures.
};
```

We will start by discussing the member variables of this class and then we will finally wrap up our coverage of the Action Animation System by examining the code to the member functions.

**TCHAR m_strName[128]**

This member will hold the name of the action as read in from the .ACT file. In the example .ACT file we looked at earlier, the first action was called 'Walk_At_Ease'. This is the name that would be stored in this member for that action. This value will be set when the CActionDefinition::LoadAction method is called and the name of the current action being loaded is extracted from the file.

Notice in the above code that there is an inline method called GetName which will return the name of the action. We saw this function being used in the CActor::ApplyAction method earlier. It searched through all of the actions using the GetName method to test against the name of the requested action passed by the application. If a match was found, this was assumed to be the action that the application would like to apply and its ApplyAction method was called. We will look at the CActionDefinition::ApplyAction method in a moment.

**USHORT m_nSetDefinitionCount**
This member contains the number of SetDefinitions that exist for this action. In our example .ACT file, each action contained three set definitions. Recall that a set definition is really just the name of an animation set that is used to comprise the action and any associated properties that should be associated with that set (such as should it transition in or not). Every action can have a different number of set definitions, so one action may involve setting a single animation set while another action may set many.

This member contains the number of animation sets that will need to be set on the mixer when this action is applied. This also describes the number of elements in the m_pSetDefinitions array (discussed below) which is an array that contains the information for each set definition.

**SetDefinition m_pSetDefinitions**
As discussed above, each action is usually comprised of several set definitions. Each set definition describes the name and properties of an animation set that should be assigned to the mixer to help achieve the action when it is applied. The number of elements in this array will be equal to the number of set definitions defined in the .ACT file for the current action. This array will be allocated when the action is loaded.

The name and properties of each animation set used by this action are stored in this array as a SetDefinition structure. Let us now describe the members of the SetDefinition structures that comprise this array.

> **TCHAR strSetName[128]**
> This contains the name of the animation set that this set definition is associated with. You will recall that when you defined a set definition in the .ACT file you had to specify the first property as the name of the animation set you would like these properties to be linked to. The animation set you name here will be one (of potentially many) that will be assigned to the animation mixer when this action is applied.
>
> **TCHAR strGroupName[128]**
> Here we store the name of the group this set definition is assigned to. This is pretty important when blending is being used since we will often want an animation set to transition from another animation set with the same group name when the action is applied. As discussed earlier, you should never have more than one set definition within a single action with the same group name.
>
> **float fWeight**
> The initial weight that you would like the animations set to have. For an animation set that is configured to fade-in, this is the target weight that the animation set's track will reach at the end of the fade-in process. For non fade-in sets, the track weight will be assigned this value the moment the set is activated.
>
> **float fSpeed**
> The speed at which the animation set should play. You will usually set this to 1.0 so that the animation plays at the default speed intended by the asset creator.

**ActBlendMode BlendMode**
This set definition property is used to determine whether the animation set should be played immediately at full weight (disabling any animation set currently playing from the same group) or whether it should slowly transition in. It will be set to one of the following members of the ActBlendMode enumerated type. This enumeration is defined within the CActionDefinition namespace in CActor.h. As you can see, it mirrors the two values we could use in the .ACT file to specify the blend mode.

```
enum ActBlendMode
{
        Off             = 0,
        MixFadeGroup    = 1
};
```

**Off**
When set to Off, no blending occurs. When the action is applied, any previous animation sets (which have not already been put into a transition out state) from the same group will be immediately halted and replaced with this new set.

**MixFadeGroup**
When the action is applied, any animation set currently playing from the previous action with a matching group name will be faded out over a period of time and the strength of this set will be faded in over that same period of time. This causes a smooth transition between animation sets in the same groups; animation sets that animate the same portions of the character for example.

**ActTimeMode TimeMode**
The TimeMode property of a set definition is something else we discussed when covering the .ACT file specification. The CActionDefinition::LoadAction will set the TimeMode property of each of its set definitions to one of three members of the ActTimeMode enumerated type (depending on which of these values is specified in the .ACT file).

```
enum ActTimeMode
{
        Begin               = 0,
        MatchGroup          = 1,
        MatchSpecifiedGroup = 2
};
```

**Begin**
The animation set will start from the beginning when played. That is, the track timer to which it is assigned will be reset. This is used mainly if the animation set you are playing does not have to be synchronized either to another animation set that is currently animating another part of the actor or if you do not need this animation set to smoothly take its periodic position from an animation set in the same group that is already playing from a previous action.

**MatchGroup**
In this mode, when the animation set is assigned to the track, its track time will be copied from the track of an animation set in the same group that was being used by the previous action.

A good example of a need for this mode would be if two actions contain the exact same animation set for the 'Legs' group. Both actions may have different animation sets for the torso but may both use the 'Walk' animation set for the legs. If action 1 was currently being played and the player was halfway through a stride, we would not want the animation set working the legs to have its periodic position snapped back to zero when the second action was applied. Instead, we would want the animation set working on the legs in the second action to start at a periodic position that continues the stride of the character and takes over from where the 'Legs' animation set in the previous action left off.

**MatchSpecifiedGroup**
This mode is useful for synchronizing the different animation sets together. In out example .ACT file, the torso animation for the 'Walk_At_Ease' action uses this mode so that it can take its timing from the 'Legs' group. That way the legs and arms are synchronized as the character walks.

**float fMixLength**
If the blend mode of this SetDefinition has been set to MixFadeGroup, then this value will contain the number of seconds this animation set will take to transition to full strength and how long any animation set from a previous action in the same group will take to have its strength completely faded out.

If the BlendMode is set to 'Off', this property will not be used.

**TCHAR strTimeGroup[128]**
If the TimeMode of the SetDefinition has been set to MatchSpecifiedGroup then this member will contain the name of the group that this animation set will take its track time from when first applied to the mixer. For example, a SetDefinition which has been assigned to the 'Arms' group could have its TimeMode set to take its initial starting position from the 'Legs' group animation set that is currently playing.

If TimeMode is not set to MatchSpecifiedGroup, this property is not used.

**bool bNewTrack**
**double fNewTime**
These final two members of the SetDefinition structure are used for temporary data storage during the processing of an action inside the CActionDefinition::ApplyAction method. We will see them being used later.

We have now discussed all the member variables and structures used by CActionDefinition. Let us now look at its member functions.

## CActionDefinition::CActionDefinition

The constructor places the terminating null character in the first position of the name string (empty string) and sets the definition count to zero. It also sets the definition set array pointer to NULL. These variables will not be populated with meaningful values until the CActionDefinition::LoadAction method is called to populate this object.

```
CActionDefinition::CActionDefinition( )
{
    // Clear required variables
    m_strName[0]            = _T('\0');
    m_nSetDefinitionCount   = 0;
    m_pSetDefinitions       = NULL;
}
```

## CActionDefinition::~CActionDefinition

The destructor quite expectedly releases the set definitions array and sets its pointer to NULL and restores all values to their default state.

```
CActionDefinition::~CActionDefinition( )
{
    // Delete any flat arrays
    if ( m_pSetDefinitions ) delete []m_pSetDefinitions;

    // Clear required variables.
    m_strName[0]            = _T('\0');
    m_nSetDefinitionCount   = 0;
    m_pSetDefinitions       = NULL;
}
```

## CActionDefinition::LoadAction

This method is called by the CActor::LoadActionDefinitions method. You will recall when we covered that function earlier that it read the number of actions in the .ACT file and then allocated the actor's array of CActionDefinition objects. It then looped through each action and called its LoadAction function to populate it with the action data from the .ACT file.

LoadAction is a little long and rather uneventful. It just uses the Win32 .ini file reading functions to read the various properties of the action from the file and store that data in its internal members.

The function is passed two parameters by the CActor::LoadActionDefinitions function. The first is the index that will be appended to the word 'Action' to describe the Action block in the .ACT file that contains the data for this action definition. The second parameter is the name (and path) of the .ACT file that we are currently in the process of fetching the action information from.

The first thing this function does is build a string (using the _stprintf function) containing the name of the data block in the .ACT file we wish to access. Once we have this string constructed and stored in the local variable (strSectionName) we flush any data that may have previously existed in this action definition object. We release its SetDefinitions array if it exists and make sure that its definition count is set to zero at the start of the loading process.

```
HRESULT CActionDefinition::LoadAction( ULONG nActionIndex, LPCTSTR ActFileName )
{
    TCHAR strSectionName[128];
    TCHAR strKeyName[128];
    TCHAR strBuffer[128];
    ULONG i;

    // First build the action section name
    _stprintf( strSectionName, _T("Action%i"), nActionIndex );

    // Remove any previous set definitions
    if ( m_pSetDefinitions ) delete []m_pSetDefinitions;
    m_pSetDefinitions     = NULL;
    m_nSetDefinitionCount = 0;
```

The first property we fetch is the DefinitionCount. Every action in the .ACT should have a definition count property which describes how many set definitions comprise the action. This also tells us how large we should allocate the m_pSetDefinitions array for this action.

As before, because the value we wish to read is an integer value, we use the Win32 function designed for the task of reading integer values from an .ini file (GetPrivateProfileInt).

```
    // Retrieve the set definition count
    m_nSetDefinitionCount = ::GetPrivateProfileInt(  strSectionName,
                                                     _T("DefinitionCount"),
                                                     0,
                                                     ActFileName );

    // If there are no sets, we were most likely unable to find the action
    if ( m_nSetDefinitionCount == 0 ) return D3DERR_NOTFOUND;

    // Allocate the setdefinition array
    m_pSetDefinitions = new SetDefinition[ m_nSetDefinitionCount ];
    if ( !m_pSetDefinitions ) return E_OUTOFMEMORY;
```

The first parameter to GetPrivateProfileInt is the string we built at the start of the function. It contains the name of the action block in the .ACT file we wish to read data from. As the second parameter we pass a string describing the property we wish to read within this action block. In this call we want to fetch the value of the DefinitionCount property. You must type property names exactly as you have defined them in the .ACT file. As the third parameter we always pass in a default value that will be used if the property cannot be found in the file or in the specified block/section. In this case, if the Definition Count property for an action is missing, this is considered a corrupt action description and we assign the value of zero. This will cause the function to exit on the very next line and the action will not be loaded. As the final parameter to this function we pass the name of the .ACT file we wish to read.

On function return, we should have the set definition count in the member variable m_nSetDefinitionCount and if this value is zero, we return from the function and do not load this action. Otherwise, we use this value to allocate an array of SetDefinition structures of the correct size.

We will now fetch the name of the action. Since the name of the action is a string and not an integer value we will use the Win32 GetPrivateProfileString function to fetch this data. The function works in an almost identical way but takes extra parameters. The first parameter is the block name in the .ACT file we wish to pull the values from. This is the current Action block being read. As the second parameter we pass the name of the property within this block that we wish to read. In this next call, we wish to read the value of the Name property for this action. As the third parameter we pass in the string that should be returned if the property is not found; in this case we simply specify an empty string. As the fourth parameter we pass in the string variable that will contain the value on function return. We wish to store the name of the action in the m_strName member variable. The fifth parameter is the number of character we would like to read. This is to make sure that we do not overflow the memory bounds of our string. We read the first 127 characters, since an action name is not likely to ever be longer than this and our m_strName member variable only has space for 128 characters (i.e., 127 characters plus the terminating NULL).

```
    // Retrieve the name of the action
    ::GetPrivateProfileString( strSectionName,
                               _T("Name"),
                               _T(""),
                               m_strName,
                               127,
                               ActFileName );
```

At this point we have the name of the action stored, the number of set definitions it should contain and we have allocated the SetDefinitions array. Let us now loop through each element in that array and populate each set definition with data from the file.

In the first section of the loop (shown below) we get a pointer to the current element in the SetDefinitions array. First we read the name of the animation set and the name of the group associated with this definition and store them in the definition set's strSetName and strGroupName members.

```
    // Loop through each set defined in the file
    for ( i = 0; i < m_nSetDefinitionCount; ++i )
    {
        // Retrieve a pointer to the definition for easy access
        SetDefinition * pDefinition = &m_pSetDefinitions[i];

        // Get the set name
        _stprintf( strKeyName, _T("Definition[%i].SetName"), i );
        ::GetPrivateProfileString( strSectionName,
                                   strKeyName,
                                   _T(""),
                                   pDefinition->strSetName,
                                   127,
                                   ActFileName );

        // Get the group name
```

```
        _stprintf( strKeyName, _T("Definition[%i].GroupName"), i );
        ::GetPrivateProfileString(  strSectionName,
                                    strKeyName,
                                    _T(""),
                                    pDefinition->strGroupName,
                                    127,
                                    ActFileName );
```

Notice that before we access each property, we build a string containing the name of the property we wish to fetch. For example, if we are fetching the animation set name of the third set definition, the name of this property inside the action's data block would be 'Definition3.SetName'. Similarly, the group name would be assigned to the property 'Definition3.GroupName'. Open up the .ACT file and examine it for yourself so you can follow along with this function.

In the next section of code we need to determine if this set definition is set to blend or not. The blend mode of this definition will be set to either 'Off' or 'MixFadeGroup'. Notice in the following function call to GetPrivateProfileString that a default string of 'Off' is returned if no matching property has been specified in the .ACT file. This means if no blend mode is specified, it is equivalent to specifying no blending.

```
        // Get the blend mode flags
        _stprintf( strKeyName, _T("Definition[%i].BlendMode"), i );
        ::GetPrivateProfileString(  strSectionName,
                                    strKeyName,
                                    _T("Off"),
                                    strBuffer,
                                    127,
                                    ActFileName );


        if ( _tcsicmp( strBuffer, _T("MixFadeGroup") ) == 0 )
            pDefinition->BlendMode = ActBlendMode::MixFadeGroup;
        else
            pDefinition->BlendMode = ActBlendMode::Off;
```

Once we have fetched the blend mode string we test its value and set the definition's BlendMode member to the correct enumeration.

We will now read the TimeMode of the set definition. It can be either 'Begin', 'MatchGroup' or MatchSpecifiedGroup. A default string of 'Begin' will be returned if no TimeMode property exists in the .ACT file for the current set definition being read. In other words, if no time mode property exists in the .ACT file for a given set definition, by default time the animation set will be started from the beginning whenever the action is applied.

```
        // Get the time mode flags
        _stprintf( strKeyName, _T("Definition[%i].TimeMode"), i );
        ::GetPrivateProfileString(  strSectionName,
                                    strKeyName,
                                    _T("Begin"),
                                    strBuffer,
                                    127,
```

```
                                    ActFileName );

        if ( _tcsicmp( strBuffer, _T("MatchGroup") ) == 0 )
            pDefinition->TimeMode = ActTimeMode::MatchGroup;
        else
            if ( _tcsicmp( strBuffer, _T("MatchSpecifiedGroup") ) == 0 )
            pDefinition->TimeMode = ActTimeMode::MatchSpecifiedGroup;
        else
            pDefinition->TimeMode = ActTimeMode::Begin;
```

After we have retrieved the string from the file we test its contents and set the definition's TimeMode member to the correct member of the ActTimeMode enumeration.

In the next section we retrieve the Weight and Speed properties for this set definition. Since these are floating point values and Win32 exposes no GetPrivateProfileFloat function, we will have to read the values as strings and then convert the strings into floats using the _stscanf function.

You can see in the following code that after we extract the weight value (with a default weight of 1.0 if the property is not found in the file), we have ourselves a floating point value stored in the string strBuffer. We then use the _stscanf function to fetch the float from that string and store it in the fWeight member for the current definition. The first parameter to the _stscanf is the string we wish to extract the values from and the second parameter is a format string that informs the function how many values we wish to extract and what their types are. Using the '%g' string tells the function that we wish to extract one floating point value from the beginning of the string. Since we are only extracting one value, there is only one variable specified after this parameter -- the address of the variable we would like this value copied into. We perform the same procedure to extract the Speed property as well.

```
        // Retrieve and scan the weight float var
        _stprintf( strKeyName, _T("Definition[%i].Weight"), i );
        ::GetPrivateProfileString(  strSectionName,
                                    strKeyName,
                                    _T("1.0"),
                                    strBuffer,
                                    127,
                                    ActFileName );

        _stscanf( strBuffer, _T("%g"), &pDefinition->fWeight );

        // Retrieve and scan the speed float var
        _stprintf( strKeyName, _T("Definition[%i].Speed"), i );
        ::GetPrivateProfileString(  strSectionName,
                                    strKeyName,
                                    _T("1.0"),
                                    strBuffer,
                                    127,
                                    ActFileName );

        _stscanf( strBuffer, _T("%g"), &pDefinition->fSpeed );
```

At this point we have read in the standard properties that are applicable in all circumstances, but we now have to read two optional ones. For example, we know that if the BlendMode of this set definition is set

to MixFadeGroup then there will also be a MixLength property specified for this set which describes how long the transition should take to perform. This is a float value that specifies this time in seconds so once again, we fetch the value as a string and scan it into the set definition's fMixLength member.

```
        // If we're blending, there will be an additional parameter
        if ( pDefinition->BlendMode == ActBlendMode::MixFadeGroup )
        {
            // Retrieve and scan the various floats
            _stprintf( strKeyName, _T("Definition[%i].MixLength"), i );
            ::GetPrivateProfileString(    strSectionName,
                                          strKeyName,
                                          _T("0.0"),
                                          strBuffer,
                                          127,
                                          ActFileName );

            _stscanf( strBuffer, _T("%g"), &pDefinition->fMixLength );

            // If we returned a zero value, turn off mix fading of the group
            if ( pDefinition->fMixLength == 0.0f )
                  pDefinition->BlendMode = ActBlendMode::Off;

        } // End if fading
```

Notice that if no MixLength property is found, a default of zero is returned. This causes the definition's BlendMode member to be set to ActBlendMode::Off (i.e., blending is disabled).

The last value we read in for the current definition is the TimeGroup property. It will only be specified in the .ACT file if the definition has had its TimeMode set to MatchSpecifiedGroup. Recall that this means when the animation set is assigned to the mixer, it will have its track time set to the position of an existing track in the specified group. This property contains the name of that group.

```
        // If we're timing against a specified group, get additional parameter
        if ( pDefinition->TimeMode == ActTimeMode::MatchSpecifiedGroup )
        {
            // Retrieve and scan the various floats
            _stprintf( strKeyName, _T("Definition[%i].TimeGroup"), i );
            ::GetPrivateProfileString(    strSectionName,
                                          strKeyName,
                                          pDefinition->strGroupName,
                                          pDefinition->strTimeGroup,
                                          127,
                                          ActFileName );

        } // End if matching against alternate group

    } // Next new set definition

    // Success!!
    return S_OK;
}
```

That is the end of the loop and the end of the function. This loop is executed for each definition in the current action. At the end, the CActionDefinition object will have a fully populated SetDefinition array

where each element can be used by the ApplyAction method to setup the animation sets correctly when the action is applied.

## CActionDefinition::ApplyAction

This is the function that provides the core processing of our Action Animation System. Not only does it set the animation sets for the action on the animation mixer, it also performs fade outs of the animation sets from a previous action. Additionally, it keeps the actor's CActionStatus object up to date so that we always know which tracks are in use, which ones are transitioning out, and which ones are free for use.

The function can appear intimidating due to its sheer size. However once you take a closer look, you should see that it is really not so difficult. The code is really repeating the same steps for both the blending and non-blending cases, so it should be pretty easy to grasp once you understand the main logic.

This function is called by the CActor::ApplyAction function after it has found the CActionDefinition object with a name that matches the one requested by the application. Once the correct action is found, its ApplyAction method is called to configure the animation mixer. Let us look at that code now.

The first thing this function does is get a pointer to the actor's CActionStatus object. As this contains the current state of each track (is it in use or not for example) we will definitely need this information.

```
HRESULT CActionDefinition::ApplyAction( CActor * pActor )
{
    SetDefinition                   * pDefinition;
    CActionStatus                   * pStatus;
    CActionStatus::TrackActionStatus * pTrackStatus;
    ULONG                             i, j, nCount;
    D3DXTRACK_DESC                    TrackDesc;

    // Retrieve the actors action status
    pStatus = pActor->GetActionStatus();
    if ( !pStatus ) return D3DERR_INVALIDCALL;
```

This step is very important for finding tracks that have fully transitioned out and updating the relevant track properties in the CActionStatus object to reflect that this track is now free.

We said earlier when discussing how this system will work, that if a given set definition is configured to blend, then its weight will transition in over a period of time and a previous animation set that was playing from the same group will transition out over the same period. This produces a smooth blend from the old set to the new one. The fading out of the set is done by setting an event on the previous animation set's track that will reduce its weight to zero over this period. We then set another event on that track that will be triggered when the weight reaches zero which will disable the track. If a track has been disabled since we last applied any actions, we need to know about it so that we can update the track status in our CActionDefinition object. If we do not do this, our system will think the track is still in use and we will find ourselves running out of tracks in a very short time as actions are repeatedly applied.

The next section of code performs this synchronization process between the tracks on the animation mixer and the TrackActionStatus structures in our CActionStatus object. Let us first look at the code and then we will discuss it.

```
    // Mark disabled tracks as free for use.
    nCount = pActor->GetMaxNumTracks();

    for ( j = 0; j < nCount; ++j )
    {
        // Retrieve action status information for this track.
        pTrackStatus = pStatus->GetTrackStatus( j );

        // Get the current track description.
        pActor->GetTrackDesc( j, &TrackDesc );

        // Flag as unused if the track was disabled
        if ( TrackDesc.Enable == FALSE )
        {
            pTrackStatus->bInUse            = false;
            pTrackStatus->bTransitioningOut = false;

        } // End if track is disabled
    } // Next track
```

We first fetch the number of tracks on the animation mixer and set up a loop to process each track. For each track, we get a pointer to the associated TrackActionStatus structure in our CActionStatus object and also call the actor's GetTrackDesc method to fetch a D3DXTRACK_DESC containing the properties of the actual mixer track. Then we test to see if the track on the mixer has been disabled. If it has, the Enable member of the D3DXTRACK_DESC structure will be set to false. As you can see, if it is, it means that this track is no longer being used by our action system and we should set the bInUse and bTransitioningOut booleans of the corresponding track on our CActionStatus object to false. We will now recognize that this track is available for use by our system. All we are really doing is performing garbage collection and making sure we recycle any tracks that were being used for transitioning out but have since been disabled by a sequencer event that occurred when the weight of the track reached zero.

At this point, our CActionStatus object correctly reflects the current state of the tracks with respect to availability. Now it is time to loop through each of the set definitions in this action and figure out which track the definition should be assigned to, and what its starting position and transition status should be. With the exception of a small piece of code at the bottom, most of the logic in this function is contained inside this loop. That is, all the code you are about to see is carried out for each set definition in the action. Remember, a set definition represents an animation set and its associated properties.

```
    // Loop through each of our set definitions
    for ( i = 0; i < m_nSetDefinitionCount; ++i )
    {
        // Retrieve the definition pointer for easy access
        pDefinition = &m_pSetDefinitions[i];

        // Reset sensitive variables
        pDefinition->bNewTrack = false;
        pDefinition->fNewTime  = 0.0f;
```

The first thing we do at the head of this loop (above) is get a pointer to the current set definition we are processing. From this point on, pDefinition will be the pointer we use for ease of access.

We also reset the bNewTrack and fNewTime variables to false and zero respectively. You may recall when we discussed the members of the SetDefinition structure that these two members were described as temporary storage variables. They were not populated by the loading process with values from the .ACT file but will be used in this function to record whether this definition needs to be assigned to a new track and the starting position the animation set should play from. We set these to default values for now because we do not yet know if a new track will be required or if this animation set is currently assigned to a track already and can be recovered. We have also not yet calculated what the position of the track should be when the set is assigned. We will need to determine this in a moment based on the TimeMode that has been set for this definition. If the TimeMode of this set has been set to 'Begin' then zero will be the track position we wish it to start from.

At the end of this loop, in each definition we will have stored whether or not a new track is needed for it and how the track position should be set. We can then do a final pass through the tracks on the mixer and start our assignments. This will make a lot more sense as we progress; for now just know that we are just trying to determine the track position for each set and whether a new track is needed.

If the TimeMode for this set definition is set to 'Begin' then we have already stored the correct track position in the definition's fNewTime member. However, if one of the other time modes have been activated, we will try to find a track that is already active which has an animation set assigned to it in the same group (or the specified group) and will inherit its time to ensure a smooth takeover. The entire next section of code is executed only if this set definition's time mode is set to either MatchGroup or MatchSpecifiedGroup.

```
        // Get timing data from actor if required
        if ( pDefinition->TimeMode != ActTimeMode::Begin )
        {
            // Choose which group name to pull timing from.
            TCHAR * pGroupName = pDefinition->strGroupName;

            if ( pDefinition->TimeMode == ActTimeMode::MatchSpecifiedGroup )
                 pGroupName = pDefinition->strTimeGroup;
```

In the above code we first test to see which of the two timing match modes this set definition has been configured for. If the TimeMode being used is set to MatchGroup then the local variable pGroupName will point to the name of this definition's group. If MatchSpecifiedGroup is being used, then this same pointer is instead assigned to the value stored in the definition's str_TimeGroup member. Either way, this means at the end of this code, the local variable pGroupName contains the name of the group we wish to inherit the time from (if possible). We will now need to test each track currently assigned to the mixer and inherit its time if it belongs to the same group. After examining all this function code and the steps involved in transitioning between two actions, you will understand that there will only ever be one currently active track on the mixer with the same group name. This does not include tracks which are transitioning out since they are not considered to be currently active. As such, they are left to their own devices to fade out. Therefore we are now going to search for this track and inherit its track position.

In the next section of code we set up a loop to loop through each track on the mixer. For each track, we fetch the corresponding TrackActionStatus structure from our CActionStatus object so we can examine if the track is currently in use or not. If the track is not in use or it is in the process of transitioning out then this track is essentially not currently active and we have no interest in inheriting its track position. So we skip it. If however, the track is currently in use by the action system and has a definition assigned to it with a group name that matches the one we wish to take our timing from, we will fetch the position of that track (using our CActor::GetTrackDesc function) and will assign this position to the start time stored in the current set definition we are processing. This entire process is shown below.

```
            // Find the track in this group not currently transitioning out
            nCount = pActor->GetMaxNumTracks();
            for ( j = 0; j < nCount; ++j )
            {
                // Retrieve action status information for this track.
                pTrackStatus = pStatus->GetTrackStatus( j );

                // Skip if this track is not in
                if ( pTrackStatus->bInUse == false ||
                     pTrackStatus->bTransitioningOut == true ) continue;

                // Does this belong to this group?
                if ( _tcsicmp( pTrackStatus->strGroupName, pGroupName ) == 0 )
                {
                    // Get the current track description.
                    pActor->GetTrackDesc( j, &TrackDesc );

                    // Store the position and break
                    pDefinition->fNewTime = TrackDesc.Position;
                    break;

                } // End if matching group

            } // Next Track

        } // End if need timing data
```

At this point, we have processed the time modes and our fNewTime member stores the position that its track should be set to start at when we do eventually assign it to a track at the tail of the function. The timing processing is now all done.

In the next phase we have to figure out whether this animation set needs to be blended in and a currently active set from the same group faded out. Unfortunately, things are not quite as black and white as it might seem. If the animation set for this set definition that we wish to apply matches the name of an animation set already assigned to the mixer by a previous action and the group names match, it would make little sense to have the two exact sets, one fading out and one fading in, taking up two valuable tracks on the mixer. Therefore, this is actually going to be a two step process. If mixing is being used and an exact animation set is currently in use, we will just recover it. We will just set it to fade back in, from whatever its current strength is to whatever is desired by the set definition we are currently processing. If an exact set and group combination does not exist, then a new track will be needed for this set definition. If an animation set is assigned to another track which is part of the same group, it will be faded out.

Of course, in order to do any of this we must first know whether the animation mixer currently has an animation set name/group combo assigned to one of its tracks which is exactly the same as the one contained in the set definition we are about to apply. The next section of code performs this search.

A loop is constructed to test each TrackActionStatus structure in the CActionStatus object. For each track we find which is flagged as currently in use we will see if its group name matches the group name of the set definition we wish to apply. If it does, then we know the groups match so we will continue on to see if the animation set names match. If they do, we have found an exact match and the set definition we wish to apply will not need a new track. It can simply take over the track of this identical combination. As soon as a match is found the loop breaks and the loop variable's value is assigned to the local variable nSetMatched. This will contain the index of track on the animation mixer that the set definition we are about to apply will recover.

```
long nSetMatched = -1;

// First search for an exact match for this group / set combination
nCount = pActor->GetMaxNumTracks();
for ( j = 0; j < nCount; ++j )
{
      // Retrieve action status information for this track.
      pTrackStatus = pStatus->GetTrackStatus( j );

      // Skip if this track is not in use
      if ( pTrackStatus->bInUse == false ) continue;

      // Does this belong to this group?
      if ( _tcsicmp(    pTrackStatus->strGroupName,
                        pDefinition->strGroupName     ) == 0 )
      {
            // Is it physically the same set?
            if ( _tcsicmp( pTrackStatus->strSetName,
                           pDefinition->strSetName ) == 0 )
                  nSetMatched = (signed)j;

            if ( nSetMatched >= 0 ) break;

      } // End if matching group

} // Next Track
```

All we have determined at this point is a value for nSetMatched which will either contain the index of a track to be recovered or -1 if no set/group combination currently exists. The value of this variable is used to make decisions in the next two (very large) conditional code blocks. The first code block processes the case when the definition we are about to apply require blending and the second contains almost identical code that processes the non-blending case. Even in the non-blending case, we will still recover a track with a matching set / group instead of assigning it a new track. It would be pointless to turn off one track to enable another with the exact same animation set assigned. We may as well just keep the one we were using before. Note in the above code that even if the track is transitioning out, we will recover it and flag it as a match so that we can recover it and fade it back in later.

Let us start to look at the blending case one section at a time, as it is a rather large code block. Indeed, this code block is further divided into two conditional code blocks which are executed depending on whether we are assigning a new track or recovering a track for this definition.

```
        // Does this definition require mix blending?
        if ( pDefinition->BlendMode == ActBlendMode::MixFadeGroup )
        {
```

The next block of code is executed if, during the previous search, we found that a track on the mixer currently has the same animation set with the same group name as the definition we wish to apply.

```
            // Did we find an exact match for this group / set combination?
            if ( nSetMatched >= 0 )
            {
                // This is the same set as we're trying to blend in, first thing
                // is to kill all events currently queued for this track
                pActor->UnkeyAllTrackEvents( nSetMatched );

                // Clear status
                pTrackStatus = pStatus->GetTrackStatus( nSetMatched );
                pTrackStatus->bInUse            = true;
                pTrackStatus->bTransitioningOut = false;
                pTrackStatus->pActionDefinition = this;
                pTrackStatus->nSetDefinition    = i;
```

If we are going to take over this track with our new set definition, we must be prepared for the fact that this track may have been in the process of being transitioned out when a previous action was applied. If this is the case then it will have a sequencer event set for that track that is due to disable the track when the transitioning out is complete. We no longer wish this to happen since we are going to recover the track and start to fade it back in. Therefore, in the above code we un-key any events that were set for the track so that they are no longer executed (thus preventing disabling the track). We then update the corresponding track properties in the CActionStatus object to reflect that this track is now in use again and is not transitioning out. We also store the CActionStatus object responsible for setting up this track as well as the set definition index. This is helpful if the application wishes to query this information.

Because the track that we have recovered may have been transitioning out or may have had a different weight than the one in the new set definition we are applying, we will set a KeyTrackWeight event that will be triggered immediately (notice the pActor->GetTime function call in the parameter list). It will fade the current weight of the track either in or out to match the weight of the set definition we are processing. Notice that the duration of the fade is determined by the definition's fMixLength property (specified in the .ACT file).

```
                // Queue up a weight event, and bring us back to the blend weight
                // we requested
                pActor->KeyTrackWeight(   nSetMatched,
                                          pDefinition->fWeight,
                                          pActor->GetTime(),
                                          pDefinition->fMixLength,
                                          D3DXTRANSITION_EASEINEASEOUT );
```

At this point, we have set up the track to transition correctly into the weight for the current set definition, but we also have to set the position of the track. Just because we are recovering the track does not mean we want our new definition to inherit the current position property of the track. This depends on the time mode of the definition specified in the .ACT file. In the next section of code, we fetch the track description and set its position to the time we calculated earlier. We also set its speed to the speed value for this definition. Of course, if this definition is set to inherit the time from its group, then we will absolutely leave the track position alone since it is already correct.

```
                // Get the current track description.
                pActor->GetTrackDesc( nSetMatched, &TrackDesc );


                // We are recovering from an outbound transition, so we will
                // retain the timing from this set, rather than the one we chose
                // above, if they specified only 'MatchGroup' mode.
                if ( pDefinition->TimeMode != ActTimeMode::MatchGroup )
                     TrackDesc.Position = pDefinition->fNewTime;

                // Setup other track properties
                TrackDesc.Speed = pDefinition->fSpeed;

                // Set back to track
                pActor->SetTrackDesc( nSetMatched, &TrackDesc );

            } // End if found exact match
```

At this point we have set up the set definition and the mixer track to recover. The definition's bNewTrack member will still be set to false as we do not need to assign it a new track.

The above code concludes the code block that deals with the case when we have found a matching track/group combination for our set definition in the blending case. Just to recap, the above code essentially says the following: "If I am supposed to transition in and a track already has the correct set and group name applied to it, recover this track and fade it back in".

The section of code we will see is still inside the blending case code block but is only executed if the set definition we wish to blend in does not match an animation set name and group name already applied to a mixer track. When this is the case, we simply set the definition's bNewTrack member to true so we will know at the bottom of the function that we will need to assign this set definition to a new track on the mixer.

```
            else
            {
                // This definition is now due to be assigned to a new track
                pDefinition->bNewTrack = true;

            } // End if no exact match
```

The next section of code is executed in the blending code block regardless of whether a track was recovered or we need a new track assignment for this definition. It has the job of finding all tracks currently in use that are in the same group as the current definition and then fading them out. There

should only be one active track in the same group that is not transitioning out. We are essentially searching for this track so that we can start to fade it out.

The following code starts in a similar manner to the previous search code. It searches through all the tracks in the CActionStatus object and searches for one with the same group name that is in use and not in the process of transitioning out. If found, we un-key any events it may have scheduled and set a new event to execute immediately that will reduce its weight to zero or the period specified by the fMixLength property of the set definition we are about to apply. This will fade the old group set out over the same duration as our new set will be faded in.

```
            // Find all tracks in this group that are not currently
            // transitioning out, and fade them
            nCount = pActor->GetMaxNumTracks();
            for ( j = 0; j < nCount; ++j )
            {
                // Skip this track if it's our matched track
                if ( (signed)j == nSetMatched ) continue;

                // Retrieve action status information for this track.
                pTrackStatus = pStatus->GetTrackStatus( j );

                // Skip if this track is not in use or is transitioning out
                if ( pTrackStatus->bInUse == false ||
                     pTrackStatus->bTransitioningOut == true )
                     continue;

                // Does this belong to this group?
                if ( _tcsicmp( pTrackStatus->strGroupName,
                               pDefinition->strGroupName ) == 0 )
                {
                    // Kill any track events already setup
                    pActor->UnkeyAllTrackEvents( j );

                    // Queue up a weight event to fade out the track
                    pActor->KeyTrackWeight( j,
                                            0.0f,
                                            pActor->GetTime(),
                                            pDefinition->fMixLength,
                                            D3DXTRANSITION_EASEINEASEOUT );

                    pActor->KeyTrackEnable( j,
                                            FALSE,
                                            pActor->GetTime() +
                                                      pDefinition->fMixLength );

                    // Update status
                    pTrackStatus->bInUse            = true;
                    pTrackStatus->bTransitioningOut = true;

                } // End if matching group

            } // Next Track

        } // End if requires blending
```

And that is the end of the blending case. Notice in the above code that not only do we set an event to fade the contribution of the previous track out over a period of time, we also key another event for when that period of time has been exceeded that will disable the track on the animation mixer. So, if the set definition we are about to apply had an fMixLength value of 5 seconds, we would trigger an event immediately that would fade the old set out over a 5 second period. We also set a KeyTrackEnable event that will disable the track in 5 seconds time when the fade out is complete. We saw at the start of the code that the next time an action is applied, a loop is performed through all the tracks to find ones that are disabled (and we updated the CActionStatus object accordingly). Finally, at the bottom of the code we updated the TrackActionStatus structure for the track we are fading out to mark it as in use and currently transitioning out. This way the system will not try to use this track for anything else (other than a track recovery) until it has been disabled and once again becomes a free resource.

We have now covered the blending case for the current set definition. We recovered a track if possible or recorded that this set definition will need to be assigned to a new track. We have also found any tracks from a previous action with the same group name and have set them to transition out.

The next section of code is similar and basically does the whole thing all over again, but this time for the non-blended case. If the current set definition is not set to blend then any track currently assigned to the same group will simply be halted and the new animation set will take over. Once again, if we do find a track that contains the exact same animation set and group name, we will recover that track and use it. This code is simpler than the blended case as we do not have to transition anything in or out.

Since it is very similar to the previous code we will discuss it only briefly. First we test to see if we have found a track that can be recovered. If so, we un-key any events (it may have been transitioning out) and update its corresponding TrackActionStatus structure to identify that the track is in use and not transitioning out. As before, our structure will store the address of the CActionDefinition object that applied this definition along with the definition index.

```
        // Non - Blending Case
        else
        {
            // Did we find an exact match for this group / set combination?
            if ( nSetMatched >= 0 )
            {
                // This is the same set as we're trying to apply, first thing is
                // kill all events currently queued for this track
                pActor->UnkeyAllTrackEvents( nSetMatched );

                // Clear status
                pTrackStatus = pStatus->GetTrackStatus( nSetMatched );
                pTrackStatus->bInUse            = true;
                pTrackStatus->bTransitioningOut = false;
                pTrackStatus->pActionDefinition = this;
                pTrackStatus->nSetDefinition    = i;

                // Get the current track description.
                pActor->GetTrackDesc( nSetMatched, &TrackDesc );


                // The user specified a set which is already active, so we will
```

```
                // retain the timing from this set, rather than the one we chose
                // above, if they specified only 'MatchGroup' mode.
                if ( pDefinition->TimeMode != ActTimeMode::MatchGroup )
                     TrackDesc.Position = pDefinition->fNewTime;

                // Setup other track properties
                TrackDesc.Speed  = pDefinition->fSpeed;
                TrackDesc.Weight = pDefinition->fWeight;

                // Set back to track
                pActor->SetTrackDesc( nSetMatched, &TrackDesc );

            } // End if found exact match

            else

            {
                // This definition is now due to be assigned to a new track
                pDefinition->bNewTrack = true;

            } // End if no exact match
```

The above code shows that if we can recover the track, we set the position, weight, and speed of the track immediately (no fading in or out) to that specified by the definition we are applying. Otherwise, if no match was found, we set the definition's bNewTrack Boolean to true so that we know at the bottom of the function we will need to find a free track to assign this set definition to.

In the next section of code we are still inside the non–blending code block. It essentially turns off (no fade out) any track which belongs to the same group as the definition we are currently applying. We wish to turn off any set which currently exists in the same group and apply our new one instead.

```
            // Find all tracks in this group, and kill them dead
            nCount = pActor->GetMaxNumTracks();
            for ( j = 0; j < nCount; ++j )
            {
                // Skip this track if it's our matched track
                // if we didn't find a match, nSetMatched will still be -1)
                if ( (signed)j == nSetMatched ) continue;

                // Retrieve action status information for this track.
                pTrackStatus = pStatus->GetTrackStatus( j );

                // Skip if this track is not in use
                if ( pTrackStatus->bInUse == false ) continue;

                // Does this belong to this group?
                if ( _tcsicmp( pTrackStatus->strGroupName,
                               pDefinition->strGroupName ) == 0 )
                {
                    // Kill any track events already setup
                    pActor->UnkeyAllTrackEvents( j );
                    pActor->SetTrackEnable( j, FALSE );

                    // Update status
```

```
                    pTrackStatus->bInUse            = false;
                    pTrackStatus->bTransitioningOut = false;

                } // End if matching group

            } // Next Track

        } // End if does not require blending

    } // Next Definition
```

After the definition loop has completed execution, the following will have happened for each definition.

1. We will have searched for any tracks currently assigned to the controller which belong to the same group

2. If one was found for a given definition, we recover the animation set, and set it to blend back in (or just turn it back on immediately in the non-blending case); otherwise we flag that we need to configure a new track.

3. Blend all other tracks out (or turn them off in the non blending case) that belong to the same group.

At this point, all we need to do is loop through the SetDefinitions array of this action one more time and test which ones have their bNewTrack Boolean members set to true. These are the ones that were not recovered and need new tracks.

```
    // Loop through each of our set definitions and assign to tracks if required
    for ( i = 0; i < m_nSetDefinitionCount; ++i )
    {
        // Retrieve the definition pointer for easy access
        pDefinition = &m_pSetDefinitions[i];

        // Skip if not required to assign to new track
        if ( !pDefinition->bNewTrack ) continue;
```

If we get here then we have found a definition that requires a new track so we will loop through all the tracks in our CActionStatus object and search for the first one that is not in use. Once we find one, we record the index in the local TrackIndex variable and break from the loop.

```
        // Find a free track
        long TrackIndex = -1;
        nCount = pActor->GetMaxNumTracks();
        for ( j = 0; j < nCount; ++j )
        {
            // Retrieve action status information for this track.
            pTrackStatus = pStatus->GetTrackStatus( j );

            // Free track?
            if ( pTrackStatus->bInUse == false ){TrackIndex=(signed)j; break; }

        } // Next Track
```

Now that we know the track on the mixer we wish to assign this definition too we will set up a D3DXTRACK_DESC structure to correctly set the attributes of the track to the properties of the definition (speed, weight, position, etc). Remember, we have already stored the time we wish the track position to be set to earlier in the function. In the next section of code, notice that we set the weight member of the track descriptor to the weight specified by the definition; we will set this to zero initially in a moment if this definition is configured to transition in.

```
        // Build track description
        TrackDesc.Position = pDefinition->fNewTime;
        TrackDesc.Weight   = pDefinition->fWeight;
        TrackDesc.Speed    = pDefinition->fSpeed;
        TrackDesc.Priority = D3DXPRIORITY_LOW;
        TrackDesc.Enable   = TRUE;
```

The track descriptor is set up, so let us also update the corresponding track in our CActionStatus object so that we know it is in use and know the name of the animation set and the group name currently assigned to that track.

```
        // Update track action status
        pTrackStatus = pStatus->GetTrackStatus( TrackIndex );
        pTrackStatus->bInUse            = true;
        pTrackStatus->bTransitioningOut = false;
        pTrackStatus->pActionDefinition = this;
        pTrackStatus->nSetDefinition    = i;

        _tcscpy( pTrackStatus->strSetName, pDefinition->strSetName );
        _tcscpy( pTrackStatus->strGroupName, pDefinition->strGroupName );
```

Now let us assign the physical animation set used by this set definition to the mixer track that we have chosen. Also, if the definition is set to blend, we will set the weight member of the track descriptor to zero initially and will set a sequencer event that will transition the weight to its desired strength over the mix length period. Remember, if a track currently shares the same group name as this definition, this track was already instructed to fade out over the same period earlier in the function.

```
        // Apply animation set
        pActor->SetTrackAnimationSetByName( TrackIndex, pDefinition->strSetName );

        // If we are blending, key events and override starting weight
        if ( pDefinition->BlendMode == ActBlendMode::MixFadeGroup )
        {
            TrackDesc.Weight = 0.0f;
            pActor->KeyTrackWeight( TrackIndex,
                                    pDefinition->fWeight,
                                    pActor->GetTime(),
                                    pDefinition->fMixLength,
                                    D3DXTRANSITION_EASEINEASEOUT );

        } // End if blending

        // Set track description
        pActor->SetTrackDesc( TrackIndex, &TrackDesc );
```

```
    } // Next Definition

    // Success!!
    return S_OK;
}
```

Notice at the bottom of the definition loop that we finally set the properties of the mixer track that we have just set up using the CActor::SetTrackDesc function.

That was a pretty intimidating function on first glance but fortunately there is nothing too complex going on. Rather, there are just lots of small tasks being performed which cumulatively makes everything appear complex.

We have now covered all the code to the new action system. We now have the ability to compose complex action sequences from combinations of animation sets in simple text files and have our application apply those actions with a single function call.

Before we leave the subject entirely, we need to discuss the small changes that have had to be made to the AttachController and DetachController methods in CActor. You will recall that these methods were introduced to allow our actor to be referenced. We must now make sure that the action system will also work correctly during referencing.

## Safely Referencing Action Animating Actors

In Chapter Ten we wrote code that allowed multiple objects to reference the same actor, where each had different animation data or was at a different position on the animation timeline. This allowed us to instance the actor many times in the scene without having to load redundant copies of the actor geometry into memory. Our design solution was actually very simple. We decided that if multiple CObject's in our scene referenced the same actor, the actor would no longer own its own animation controller; instead, each object would manage its own controller. To update an object's animation, we simply attached its controller to the actor and then used the actor's methods to update it. Similarly, when rendering an object, we attached the object's controller to the actor and rendered the actor as normal. This solution works well and we will continue to use it.

The action system is also controller specific because the CActionStatus object maintains the current position of all the tracks on the controller. If we were to detach the controller and attach a different one, the CActionStatus object would be completely out of sync. Of course, we could write some function that would synchronize the CActionStatus object to a new controller whenever one is attached. While that would be easy to do, it would not be the wisest design choice. Remember, if we have multiple objects using the same actor, we want the actor's CActionStatus object to remember the settings of each reference because the controller itself will have no way of letting the action system know whether a track on the newly attached controller was in the process of fading out when it was last detached. We have essentially lost this information and our action system will not reference properly.

Of course, the solution is really rather straightforward. The CActionStatus object contains the state of a controller, and as each CObject referencing an actor owns its own controller, it should also own its own CActionStatus object in exactly the same way. All we need to do is add an extra parameter to our AttachController and DetachController methods that will allow us to pass/retrieve a CActionStatus object to/from a CActor object. That is, every time we attach an object's controller to an actor, its CActionStatus object will also be passed and will become the actor's current CActionStatus object. This way, each object owns both its own controller and the current state of that controller with respect to the action system. This allows the current state of an object's controller to persist during the attach/detach process.

## CActor::Detach Controller

This method has been updated to take an optional CActionStatus parameter. If your object does not wish to use the action system we can just call this function as we always have. However, if you do wish to retrieve the current CActionStatus object of the actor, we can pass in the address of a CActionStatus pointer which, on function return, will pointer to the CActionStatus object the actor was using before you detached it.

Our application uses this function inside the CScene::ProcessReference function during loading. If the reference is to an X file that has already been loaded into an actor, we essentially switch the actor into reference mode. We fetch the controller and the CActionStatus from the actor and store it in the object instead. Then, any future references to this same actor causes the controller and the status object to be cloned and stored in a new object. We discussed the CScene::ProcessReference function in the animation workbook when we introduced this reference system. If you consult this function in Lab Project 12.2 you will see that virtually nothing has changed. All we do now is store the CActionStatus object of the actor in the CObject in addition to the controller.

Let us have a look at the detach controller function, which has been slightly updated. The function first makes a copy of the controller pointer and then sets the actor's controller pointer to NULL. It is now going to surrender its controller reference to the caller. At this point the actor has no controller anymore. We also copy the address of the actor's CActionStatus object in the passed pointer and set the actor's CActionStatus pointer to NULL as well. The actor no longer owns a CActionStatus object either. The function then returns the pointer to the new controller. The CActionStatus object is returned via the output parameter.

```
LPD3DXANIMATIONCONTROLLER CActor::DetachController(CActionStatus** ppActionStatus)
{
    LPD3DXANIMATIONCONTROLLER pController = m_pAnimController;

    // Detach from us
    m_pAnimController = NULL;

    // If the user requested to detach action status information, copy it over
    if ( ppActionStatus )
    {
        *ppActionStatus = m_pActionStatus;
        m_pActionStatus = NULL;
```

```
    } // End if requested action status

    // Note, we would AddRef here before we returned, but also
    // release our reference, so this is a no-op. Just return the pointer.
    return pController;

}
```

## CActor::AttachController

This function also has a new parameter and a few new lines added at the bottom. The function now takes an additional third parameter allowing an action status object to be attached to the actor.

The first thing the function does is release the current action status object and controller being used by the actor. We then set the actor's controller and status object pointer to NULL.

```
void CActor::AttachController( LPD3DXANIMATIONCONTROLLER pController,
                               bool bSyncOutputs /* = true */,
                               CActionStatus * pActionStatus /* = NULL */ )
{
    // Release our current controller.
    if ( m_pAnimController ) m_pAnimController->Release();
    m_pAnimController = NULL;

    // Release our current action status (if provided).
    if ( m_pActionStatus ) m_pActionStatus->Release();
    m_pActionStatus = NULL;
```

If a new controller was passed, we will assign the address of this new controller to the actor's controller pointer and increase the reference count. If the bSyncOutputs parameter was set to true it means the caller would like the relative matrices of the hierarchy immediately updated to reflect the current position in the animation. We do this by calling AdvanceTime on the controller with a time delta of 0.0 seconds (we do not physically alter the position of the animation). The sequence of events is shown below.

```
    // Attach new controller if one is provided
    if ( pController )
    {
        // Store the new controller
        m_pAnimController = pController;

        // Add ref, we're storing a pointer to it
        m_pAnimController->AddRef();

        // Synchronize our frame matrices if requested
        if ( bSyncOutputs ) m_pAnimController->AdvanceTime( 0.0f, NULL );

    } // End if controller specified
```

In the final section of code we attach the passed CActionStatus object to the actor's member pointer and increase its reference count.

```
    // Attach new status object is one is provided
    if ( pActionStatus )
    {
        // Store the new status information
        m_pActionStatus = pActionStatus;

        // Add ref, we're storing a pointer to it
        m_pActionStatus->AddRef();

    } // End if status specified
}
```

# Using the Action Animation System

In Lab Project 12.3 we use the action animation system to animate our game character. In this section we will detail the points of communication between the application and the actor in all matters involving the use of this action system.

## Step 1: Updating CObject for Action Referencing

The first thing that has to be done is the expansion of our CObject structure to contain a CActionStatus pointer. As described in the previous section, this allows for proper actor referencing. Below we see the new CObject class as declared in 'CObject.h'.

```
class CObject
{
public:
    //-------------------------------------------------------------------------
    // Constructors & Destructors for This Class.
    //-------------------------------------------------------------------------
    CObject( CTriMesh * pMesh  );
    CObject( CActor   * pActor );
    CObject( );
    virtual ~CObject( );

    //-------------------------------------------------------------------------
    // Public Variables for This Class
    //-------------------------------------------------------------------------
    D3DXMATRIX                  m_mtxWorld;             // Objects world matrix
    CTriMesh                   *m_pMesh;                // Mesh we are instancing
    CActor                     *m_pActor;               // Actor we are instancing
    LPD3DXANIMATIONCONTROLLER   m_pAnimController;      // Controller
    CActionStatus              *m_pActionStatus;        // Action Status
};
```

## Step 2: Loading the Player Model

In our application, the player controlled character uses the action system. Unlike previous demos, our CPlayer object will have a proper animating actor attached to it instead of the placeholder cube mesh we have been using in previous demos. This means the CGameApp::BuildObjects function which is called at application startup, will now need to call another function to load the model that will be attached to our CPlayer object.

The function we use to load and create the actor for the CPlayer object is part of the CScene namespace and is called CScene::LoadPlayer. This function accepts the name of an X file to load and the name of an .ACT file to load. The third parameter is the address of the CPlayer object that will be assigned the actor as its third person object.

We see this function being called from the CGameApp::BuildObjects function after the main scene has been loaded. It is highlighted in bold at the bottom of the following listing. The rest of this function should be familiar to you.

```
bool CGameApp::BuildObjects()
{
    CD3DSettings::Settings * pSettings = m_D3DSettings.GetSettings();
    bool                     HardwareTnL = true;
    D3DCAPS9                 Caps;

    // Should we use hardware TnL ?
    if ( pSettings->VertexProcessingType == SOFTWARE_VP ) HardwareTnL = false;

    // Release previously built objects
    ReleaseObjects();

    // Retrieve device capabilities
    m_pD3D->GetDeviceCaps(    pSettings->AdapterOrdinal,
                              pSettings->DeviceType, &Caps );

    // Set up scenes rendering / initialization device
    m_Scene.SetD3DDevice( m_pD3DDevice, HardwareTnL );

    ULONG LightLimit = Caps.MaxActiveLights;

    // Load our scene data
    m_Scene.SetTextureFormat( m_TextureFormats );

    // Attempt to load from IWF.
    if (!m_Scene.LoadSceneFromIWF( m_strLastFile, LightLimit ))
    {
        // Otherwise attempt to load from X file
        if (!m_Scene.LoadSceneFromX( m_strLastFile )) return false;

    } // End if failed to load as IWF

    // Attempt to load the player object (ignore any error at this point)
    m_Scene.LoadPlayer( _T("Data\\US Ranger.x"),
```

```
                            _T("Data\\US Ranger.act"),
                            &m_Player );

    // Success!
    return true;
}
```

To understand how the player object is loaded and configured we must take a look inside the new CScene::LoadPlayer method. The code to this new function is shown below with descriptions.

## CScene::LoadPlayer

```
bool CScene::LoadPlayer(      LPCTSTR ActorFile,
                              LPCTSTR ActionFile,
                              LPCTSTR CallbackFile,
                              CPlayer * pPlayer )
{
    long nIndex = -1;

    // Add a new slot to the actor list
    nIndex = AddActor();

    // Allocate new actor
    CActor * pNewActor = new CActor;
    if ( !pNewActor ) return false;

    // Store the actor in the list
    m_pActor[ nIndex ] = pNewActor;
```

This function accepts the name of an X file to load for the player object, the name of an .ACT file to load describing the actions that can be performed by this actor, and a pointer to the CPlayer object. For the time being, we can ignore the third parameter (CallbackFile), since it will be discussed later.

We call CScene:AddActor to make space at the end of the scene's actor pointer array for a new actor. We like to store this in the scene's actor array with the rest of the scene actors so if anything goes wrong, the scene destructor will release it along with the rest of the actors it is managing. The AddActor method returns the array index where we should store our new actor pointer, so we allocate a new CActor object and store its pointer in the array at this position.

The next three lines are familiar. Before loading the actor from the X file we register the CALLBACK_ATTRIBUTE callback function, placing the actor into non-managed mode.

We also register a callback to handle the creation of callback keys during the actor's initial loading process. We will register the same function for this purpose as we did in Chapter Ten (CScene::CollectCallbacks). You will recall that when the actor has been loaded, this function will be called for each animation that was created. It gives the application an opportunity to fill an array of callback keys and return them to the actor. The actor then clones the animation sets with these callback keys registered. In Chapter Ten we used the callback system to register sound effects with the animation

sets. This function will be used for the same purpose again in this lab project, only a slightly more robust sound playing method will be used that requires some additional tweaking to the callback system.

```
// Load the actor from file
pNewActor->RegisterCallback(CActor::CALLBACK_CALLBACKKEYS,CollectCallbacks,this );
pNewActor->RegisterCallback(CActor::CALLBACK_ATTRIBUTEID,CollectAttributeID,this);

if ( FAILED(pNewActor->LoadActorFromX(  ActorFile,
                                        D3DXMESH_MANAGED,
                                        m_pD3DDevice,
                                        true,
                                        (void*)CallbackFile )) ) return false;
```

Notice that we now pass the CallbackFile pointer as a new parameter to the LoadActorFromX function. This will contain the name of another .ini file that contains the sound files we wish to play and the animation sets we wish them to apply to. The actor will pass this to its ApplyCallbacks function after it has been loaded. The ApplyCallbacks function will then call the CScene::CollectCallbacks function (the registered callback) for each animation set loaded from the X file, passing in this string. The callback key creation function can then open the .ini file and read in the sound effects for the current animation set be created. These sounds are then registered as callback keys with the animation set. This will all be discussed at the end of this section. Just know for now that the pointer in the LoadActorFromX parameter list is pumped straight through to any registered CALLBACK_CALLBACKKEYS function to aid in the creation of the callback keys.

Now that the actor has been fully loaded along with any animation data that may exist, we can load the action data from our ACT file into our new actor. We will also extend the limits of the actor's controller so that the action system has 20 mixer tracks to play with when performing its fades between actions. The CActor::SetActorLimits method is used for this purpose. It ultimately issues a call to the CActor::CActionStatus::SetMaxTrackCount method so the status object's TrackActionStatus array is resized to retain a 1:1 mapping with mixer tracks available on the controller.

```
    // Load the action definition file
    if ( FAILED(pNewActor->LoadActionDefinitions( ActionFile )) ) return false;

    // Set the actor limits to something reasonable
    if ( FAILED(pNewActor->SetActorLimits( 0, 20 )) ) return false;
```

At this point, our actor has been correctly configured and contains all its geometry, animation, and action data. We now need to assign this actor to an object. To do this we use the CScene's AddObject method to make space at the end of its CObject array for a new object pointer. Then we allocate a new CObject, store its pointer in the object array, and assign our new actor to its m_pActor member. Finally, before returning, we call the CPlayer::Set3rdPersonObject method to store a pointer to this CObject in the CPlayer. This is the object that the CPlayer::Render method will use to render itself.

```
    // Add a new slot to the object list
    nIndex = AddObject();

    // Create a new object
    CObject * pNewObject = new CObject;
```

```
    if ( !pNewObject ) return false;

    // Store the object in the list
    m_pObject[ nIndex ] = pNewObject;

    // Store the object details
    pNewObject->m_pActor = pNewActor;

    // Inform the player of this object
    pPlayer->Set3rdPersonObject( pNewObject );

    // Success!
    return true;
}
```

We have now seen the adjustments that we have had to make to our application to support the loading of the action data and the population of an actor with this action data. In the next section we will discuss how the CPlayer object has been modified to apply the relevant actions to its attached actor.

## Step 3: Applying Actions

You will hopefully recall that when we first introduced our CPlayer object in Chapter Four of Module I, the CPlayer::Update function was called to update the position of the player in response to user input.

In every iteration of our game loop, the CGameApp::ProcessInput function is called to determine if the player is pressing any keys or moving the mouse. In response to the movement keys being pressed for example, this function would call the CPlayer::Move function to update the velocity variable in the player object. If the mouse had been moved with the left button depressed, the CPlayer::Rotate method would be called to apply a rotation to the CPlayer's world matrix. However, the position of the player was not updated until the very end of that function when the CPlayer::Update method was called. It was this function that was responsible for updating the position of the player and its attached camera based on the currently set velocity of the player. Other factors came into play as well such as resistance forces working against the velocity vector.

Although we will not show the code to the CPlayer::Update function again here, one new function call has been added to it that will call a function called CPlayer::ActionUpdate. In this function, the player object has a chance to apply any actions it wishes based on user input or game events. This method will also be responsible for building the world matrix of the attached CObject that houses the actor.

Before we look at the code to the ActionUpdate method, we will discuss what the code is assumed to be doing in this example. Also note that the actual function code contained in Lab Project 12.3 may be different from what is shown next. At the time of writing it was still possible that different assets may be used in our final demo. Since the ActionUpdate function is largely dependant on the animation assets being used (e.g., the names of animation sets, etc.) the function code shown here is to be used only as an example of what an ActionUpdate function can look like.

In this example we will assume that the .ACT file contains the following five actions. Each action is comprised of three sets for the torso, hips, and legs respectively. We can forget about the 'Hips' set in each action as it essentially just undoes the pelvis rotation caused by walking.

- **Walk At Ease**
  In this action, the torso animation set adopts a typical walking sequence. The arms are swinging back and forth in tandem with the legs. The weapon is loosely held in one of the hands and is also swinging back and forth. The legs animation set is performing a typical footsteps animation. In this action, the character is certainly very relaxed and is not expecting any hostile activity.

- **Walk Ready**
  In this action the character is walking on patrol and is ready for anything. The torso animation set now has the gun held firmly to the character's chest so that he can assume the shooting pose at a moment's notice. The legs animation set is identical in all three walk actions – a standard footsteps animation.

- **Walk Shooting**
  When this action is applied the same legs animation is used as in the other two walk cases but the torso is now playing an animation set that holds the gun up to the soldier's shoulder in a targeting pose looking for enemies to fire at.

- **Idle Ready**
  This action will be played when the character is not currently walking and the fire button is not being pressed. The legs animation set simply has the legs in a stationary position so that the character is standing still. Whenever the character is idle, he will assume the ready pose with his torso (i.e., the animation set that has the character holding the gun to his chest).

- **Idle Shooting**
  This action uses the same torso animation set as the Walk Shooting action, but uses the same legs animation set as the Idle Ready action. This action is applied when the player presses the fire button (right mouse) but the character is not currently being moved at the same time. The character stands still, looking through the sight of his gun for hostile targets.

We can imagine when most of these actions will need to be applied. For example, if the player is currently moving and the fire button is pressed, then the 'Walk Shooting' action will be played. If the player stops moving but is still shooting, then the 'Idle Shooting' action will be applied. If the character is walking and his weapon has not been fired, we will apply the 'Walk At Ease' action. So when does the 'Walk Ready' action get applied?

In order to make the character feel a little more lifelike, the 'Walk Ready' action is played for five seconds after the fire button has been pressed and released. If we imagine that the character is currently moving, and we are playing the 'Walk At Ease' action, assume that the fire button is pressed and the 'Walk Shooting' animation is applied (which remains applied until the fire button is released). Instead of simply returning to the 'Walk At Ease' action as soon as the fire button is released, we apply the 'Walk Ready' action for 5 seconds. Assuming the fire button is not pressed again, after five seconds the character will drop his arms into the 'at ease' posture and start walking normally. This makes the

character seem a little more intelligent. In real life, a soldier would hardly assume an ‘At Ease’ posture immediately after firing a shot. He would probably fire and then remain in the ready position for some time, ready to fire again until he is sure that the threat has passed. So we add a little bit of that logic here just to demonstrate how you can use the AAS to accomplish different design ideas.

With these animation assets now explained, we are ready to examine the code to this function that implements the behavior described above.

The function is passed one parameter by the CPlayer::Update method describing the elapsed time in seconds between the last update and the current one. Here is the first section of the function that builds the world matrix for the CPlayer’s attached object. The player currently stores a right vector, up vector, look vector and a position vector, so building the matrix is a simple case of storing these vectors in the rows of the attached CObject’s world matrix.

```
void CPlayer::ActionUpdate( float TimeScale )
{
    ULONG i;

    // If we do not have a 3rd person object, there is nothing to do.
    if ( !m_p3rdPersonObject ) return;

    // Get the actor itself
    CActor * pActor = m_p3rdPersonObject->m_pActor;

    // Update our object's world matrix
    D3DXMATRIX * pMatrix = &m_p3rdPersonObject->m_mtxWorld;

    pMatrix->_11=m_vecRight.x; pMatrix->_21=m_vecUp.x; pMatrix->_31=m_vecLook.x;
    pMatrix->_12=m_vecRight.y; pMatrix->_22=m_vecUp.y; pMatrix->_32=m_vecLook.y;
    pMatrix->_13=m_vecRight.z; pMatrix->_23=m_vecUp.z; pMatrix->_33=m_vecLook.z;
    pMatrix->_41 = m_vecPos.x;
    pMatrix->_42 = m_vecPos.y;
    pMatrix->_43 = m_vecPos.z;
```

As you can see in the above code, if the CPlayer has no third person object attached to it we simply return. Otherwise we fetch a pointer to the object’s CActor and the object’s world matrix. We then build the world matrix for the third person object.

In the next section of code we will define some static variables.

```
    static float fReadyTime = 200.0f;
    static bool  bIncreaseReadyTime = false;
```

The fReadyTime is set to an arbitrarily high value of 200 seconds and will be used as a timer to determine how long the ‘ready’ action should remain applied before the character is put back into an ‘at ease’ action state. Whenever the fire button is pressed, fReadyTime will be set to 0 (zero seconds) and the bIncreaseReadyTime boolean will be set to false so that we will not increase this timer while the fire button is being pressed. As soon as the fire button is no longer being pressed, the boolean will be set to true and the timer will be incremented each time the function is called. As this timer is always set to zero seconds when the player fires, as soon as the player is no longer firing, we will apply the ‘ready’ pose

for five seconds (i.e., while fReadyTimer is less than 5.0). As soon as its value is incremented past five seconds we know that we have been in the ready pose long enough and should now apply the 'at ease' pose. This cycle repeats whenever the fire button is pressed and the timer is set back to zero.

We get the speed of the character by calculating the length of the CPlayer's velocity vector. You will see why this is needed in a moment. We then use the GetKeyState Win32 function to test if the left mouse button is being pressed. If so, we wish to apply a shooting action and reset the ready timer to zero.

```
    // Calculate movement speed
    float fSpeed = D3DXVec3Length( &m_vecVelocity );

    // In Shooting State?
    bool bShooting = false;
    if ( GetKeyState( VK_LBUTTON ) & 0xF0 )
    {
        // We are shooting
        bShooting          = true;

        // Setup timing states
        fReadyTime         = 0.0f;
        bIncreaseReadyTime = false;

    } // End if left button is pressed
    else
    {
        // Set our 'ready' timer going
        bIncreaseReadyTime = true;

    } // End if left button is not pressed
```

At the end of this loop, if we are pressing the fire button the timer will be reset to zero and the bIncreaseReadyTime boolean will be set to false indicating that we do not wish to increment the ready timer in this update and should leave it at zero. Otherwise, if the fire button is not being pressed the Boolean will be set to true, meaning a timer increment should happen. We also set the bShooting boolean to true if the fire button is being pressed so that we will know later in the function that an action should be applied that includes a torso shooting pose.

We now test the current value of the timer and if it is found to contain less than 5 seconds of elapsed time since being reset, we should be in the ready pose. So we set the bReadyState boolean to true. Otherwise, it is set to false. We also increment the 'ready' timer using the elapsed time passed into the function. This timer increment only happens if the fire button is not being pressed.

```
    // In ready state?
    bool bReadyState = false;
    if ( fReadyTime < 5.0f ) bReadyState = true;
    if ( bIncreaseReadyTime ) fReadyTime += TimeScale;
```

At this point we have determined what the torso of the character should look like. He is either shooting (bShooting=true), in the ready pose (bReadyState=true), or is walking at ease. Next we have to find out

what the legs of the character should be doing. There are two choices, they are either walking or standing still.

We decided to place the character into an idle legs pose if the speed of the character drops to less than 6 units per second. This is a value that you will want to tweak depending on the scale of your level, the resistance forces you apply, etc. In our case, 6 units of movement per second represents a very small amount of movement at the per frame level and will be cancelled out by friction fairly quickly if no additional force is applied. Therefore, as the player slowly comes to a halt, we place its legs into the idle pose slightly prematurely. This looks more natural than the character stopping while his legs are still spread apart in a walk pose, and then having the legs slide into the idle pose on the spot.

```
    // Idling?
    bool bIdle = false;
    if ( fSpeed < 6.0f ) bIdle = true;
```

We now know whether the legs are idle or not, so we can determine which action to play. If our legs are idle and we are shooting, we should apply the 'Idle1_Shooting' action. If the fire button is not being pressed but the legs are still idle we should play the 'Idle1_Ready' action.

```
    // Apply the correct action
    if ( bIdle )
    {
        // What is our pose?
        if ( bShooting )
            pActor->ApplyAction( _T("Idle1_Shooting") );
        else
            pActor->ApplyAction( _T("Idle1_Ready") );

    } // End if idling
```

That takes care of the idle case. Next we have the code block that deals with the non-idle case which surrounds the remaining code in this function. If our legs are not idle and we are shooting, we will play the 'Walk_Shooting' action. If the fire button is not pressed but the 'ready' timer contains a value of less than 5 seconds, we will apply the 'Walk_Ready' action. Otherwise, we will play the 'Walk_At_Ease' action.

```
    else
    {
        // What is our pose?
        if ( bShooting )
            pActor->ApplyAction( _T("Walk_Shooting") );
        else
            if ( bReadyState )
            pActor->ApplyAction( _T("Walk_Ready") );
        else
            pActor->ApplyAction( _T("Walk_At_Ease") );
```

At this point we have applied the correct walking action to the actor, but we should really adjust the speed of the tracks used by this action so that it matches the speed of the character. It would look very

strange if the legs and arms of the character animated at a constant speed without accounting for the actual speed of the character's forward motion.

We will calculate the animation speed by the taking the dot product between the velocity vector and the look vector of the player since this describes to us how much of the velocity is attributed to the fact that the player is walking forward. If the velocity and look vectors are aligned, a value of 1 will be returned. This will become smaller as the vectors become misaligned. Once we have the value of the dot product (which is the cosine of the angle scaled by the length of the velocity vector), we scale it by 0.027. Finding this scaling value was just a trial and error process and you may wish to change it to suit your scene scale.

```
     // Calculate the speed of the tracks
     float fDot   = D3DXVec3Dot( &m_vecLook, &m_vecVelocity );
     float fSpeed = fDot * 0.027f;
```

Finally, we will loop through each track in the actor's CActionStatus object and examine the TrackActionStatus structure. This will tell us if the track is currently in use and the action that assigned that animation set to the mixer track. We are searching for all tracks that are currently in use and that were set up by the current CActionDefinition object that we just applied. If one is found, we set the speed of that track on the mixer to the speed value just calculated.

```
     CActionStatus * pStatus = pActor->GetActionStatus();
     CActionDefinition * pDefinition = pStatus->GetCurrentAction();

      for ( i = 0; i < pStatus->GetMaxTrackCount(); ++i )
      {
       CActionStatus::TrackActionStatus *pTrackStatus=pStatus->GetTrackStatus(i);

       // Skip if this track is not in use
       if ( !pTrackStatus->bInUse ) continue;

       // Skip if this track wasn't assigned by the currently active definition
       if ( pTrackStatus->pActionDefinition != pDefinition ) continue;

       // Set the track speed
       pActor->SetTrackSpeed( i, fSpeed );

      } // Next Track

   } // End if walking
}
```

Since we are looping through all the tracks, we are guaranteed to find those that were set up by the currently applied action. This means we will update the speed of the torso, hips, and legs tracks for the current action simultaneously ensuring that the sets within a given action remain synchronized.

We have now covered the action system in its entirety. We have not only covered the code to the action system itself, but we have also examined how the application uses it to animate a game character. As mentioned, the actual contents of the CPlayer::ActionUpdate method may be quite different from the one shown above. Indeed, this is a function that will probably have its contents changed quite frequently

to suit the assets and animation sets that you use in your own game projects. When we cover scripting in Module III, much of this code will go away and we can configure animation changes offline without having to update and recompile source code. But that is for another day.

The remainder of this workbook will be spent detailing other upgrades we have made to our application framework which are not specifically related to the Action Animation System.

# Adding Multiple Sound Effect Call-Back Keys

In Chapter Ten we implemented a lab project that had sound effects that were triggered by the actor's animation sets via the callback system. The problem is, when the callback handler played those sounds in response to a callback event being triggered, it used the Win32 'PlaySound' function. To be sure, this is not a course on sound and we do not want to get caught up in the specifics of the DirectSound API (which we are going to use), but the Win32 PlaySound function is simply too limiting to continue using it moving forward. The biggest issue is that it only allows us to play one sound at a time. That means, if we wish to play the sound of our characters footsteps as he walks and we wish to have an ambient background sound, such as blowing wind, we are out of luck. This is far too restrictive, so we will also use this lab project to introduce a new way to play sounds, via two very helpful classes that ship with the DirectX SDK framework.

## Using CSoundManager

If you look in the "/Sample/C++/Common/" folder of your DirectX 9 installation directory you will see two files named DXUTSound.h and DXUTSound.cpp. These files contain the implementation of a class called CSoundManager that we can include in our projects to easily play (multiple) sounds. This class and the class it uses to represent individual sounds (CSound) wrap the DirectSound API, making it delightfully easy to load and play sound. Although the CSoundManager class exposes many functions to perform techniques like 3D positional sound, at the moment we are only interested in using this class to load and play simple sound. As such, we will only use a very small subset of the functions available.

So that we do not get caught up in the code to these classes (which are way off topic in a graphics course) we have compiled them into a .lib file which is included with this project. These files are called libSound.lib and libSound.h and they are located in the Libs directory of the project. CGameApp.h and CScene.cpp both include this header file, giving them access to the CSoundManager class.

```
#include "..\\Libs\\libSound.h"
```

The CSoundManager object is used to create CSound objects that can be played. As such, we have made it a member variable of the CGameApp class.

```
CSoundManager      m_SoundManager;
CSoundManager      * GetSoundManager     ()       { return &m_SoundManager; }
```

Notice that we have also added a new inline member function called GetSoundManager that returns a pointer to the sound manager object. This allows other modules (such as CScene for example) to retrieve the application's sound manager object and use it to create new sounds during the registration of callback key data with the actor.

As we know, the CGameApp::InitInstance method is called when the application is first started. In turn it calls a number of functions to configure the state of the application prior to the commencement of the main game loop. This method calls functions such as CreateDisplay and TestDeviceCaps and the BuildObjects method which actually loads the geometry. Since the CSoundManager object requires some initialization of its own, we have added a new method to the CGameApp class called CreateAudio. Below we show how this function is called from the InitInstance function. The new line is highlighted in bold.

```
bool CGameApp::InitInstance( HANDLE hInstance, LPCTSTR lpCmdLine, int iCmdShow )
{
    // Store the initial starting scene
    m_strLastFile = _tcsdup( _T("Data\\Landscape.iwf") );

    // Create the primary display device
    if (!CreateDisplay()) { ShutDown(); return false; }

    // Create the primary audio device.
    if (!CreateAudio()) { ShutDown(); return false; }

    // Test the device capabilities.
    if (!TestDeviceCaps( )) { ShutDown(); return false; }

    // Build Objects
    if (!BuildObjects()) { ShutDown(); return false; }

    // Set up all required game states
    SetupGameState();

    // Setup our rendering environment
    SetupRenderStates();

    // Success!
    return true;
}
```

The CreateAudio method contains only three lines of code to call methods of the CSoundManager object to initialize it. This involves informing the operating system of the control we would like over the sound hardware and the format in which we would like sounds to be played. Here is the code to the function.

```
bool CGameApp::CreateAudio()
{
    // Create the audio devices.
    if( FAILED(m_SoundManager.Initialize( m_hWnd, DSSCL_PRIORITY ))) return false;
    if( FAILED(m_SoundManager.SetPrimaryBufferFormat( 2 ,22050,16))) return false;

    // Success!
    return true;
}
```

In the first line we call the CSoundManager::Initialize function, passing in the application's main window handle and the flag DSSCL_PRIORITY. When using a DirectSound device with the priority cooperative level (DSSCL_PRIORITY), the application has first rights to audio hardware resources (e.g., hardware mixing) and can set the format of the primary sound buffer. Game applications should use the priority cooperative level in almost all circumstances. This level gives the most robust behavior while allowing the application control over sampling rate and bit depth.

In the second line we have to inform the sound manager of the format of the sounds we would like it to create and play in its primary buffer. We can think of the primary buffer as being very much like the audio equivalent of a 3D device's frame buffer. It is where all of the individual sounds that are currently playing get blended together to produce the final audio output.

As the first parameter to the CSoundManager::SetPrimaryBufferFormat method, we pass in the number of channels we would like the primary buffer to have. We pass in a value of 2 because we would like it to play sounds in stereo. A stereo sample has two separate channels of audio within the same .wav file (it has separate audio streams for the left and right speakers). In short, if we have stereo sound samples that we wish to load and play, this will ensure that the DirectSound device will be able to reproduce the final stereo output. If you wanted to play only mono sounds, you could pass 1 for this parameter. A mono sound has only one channel of audio data which gets duplicated in each speaker.

As the second parameter we pass in a sample rate of 22050, which means we are setting the quality at which it plays back sounds to 22khz (pretty fair quality). The third parameter specifies that we would like the primary buffer to be able to play back sounds in 16-bit sound quality. 16-bit sound has much nicer audio quality then 8-bit sounds. Of course, if your sound samples have been recorded at 8-bit to begin with, then they are still going to sound bad on a 16-bit quality device. However, this setting means that 16-bit sound samples will not have to be played back at 8-bit resolution.

When the above function returns, the sound manager will be ready for action and the application can use it to load and create sounds. From this point on, the only method we will be interested in calling is the CSoundManager::Create method. This method is defined as shown below. Actually, the function has many optional parameters after the two shown here, but these are the only two we are going to use, so we will worry about them. This function is defined in DXUTSound.h.

**HRESULT Create( CSound** ppSound, LPWSTR strWaveFileName )**

The first parameter is the address of a CSound object pointer which will point to a valid CSound object on function return. The second parameter is a string containing the name of the wav file we would like to load. When the function returns, the CSound object will encapsulate that sound effect and we can use its CSound::Play method to play the sound. We can create as many CSound objects as we want just by calling this function over and over again for the different sound effects we wish to load.

Below, we see an example of creating a CSound object for a wav file called 'footsteps.wav'.

```
CSound * pSound = NULL;
m_pSoundManager->Create( &pSound, "footsteps.wav" );
```

At this point the sound would be loaded and would be ready to play. Whenever we wish to play that sound we can simply do the following:

```
pSound->Play( )
```

As you can see, it is delightfully easy to use. If we play a sound and a sound is already playing, they will be mixed in the primary buffer of the DirectSound device and both sounds will heard simultaneously. In fact, there is no hard limit on the number of sounds that can be played simultaneously, although you will see a drop in game performance if you try to play too many. When multiple sounds are being played, they all have to be mixed in the primary buffer, so there is some associated overhead.

We can also pass parameters to the CSound::Play method to tell it how we would like it to play. For example, we might specify that we would like the sound to play repeatedly (i.e., loop). The code snippet shown below comes from the very bottom of the CScene::LoadSceneFromIWF function in Lab Project 12.3. It creates and plays a looping wind sound effect while the application is running. Notice how we query the CGameApp class for its CSoundManager pointer and then use it to load a wave file called 'Wind Ambient.wav'. We then play the sound immediately.

**Excerpt from CScene.cpp : LoadSceneFromIWF**

```
CSoundManager * pSoundManager = GetGameApp()->GetSoundManager();
if ( pSoundManager )
{
  // Release the old sound if it's resident
  if ( m_pAmbientSound ) delete m_pAmbientSound;

  // Create the ambient sound
  pSoundManager->Create(&m_pAmbientSound,_T("Data\\Sounds\\Wind Ambient.wav"));
  m_pAmbientSound->Play( 0, DSBPLAY_LOOPING );

} // End if application has sound manager.
```

Notice that this time we are passing in two parameters to the Play function. The first is the priority of the sound. We pass in the lowest (default) priority of 0, stating that this sound is no more important than any other sound our application might play. The second parameter is a flag that describes how the DirectSound buffer (all sounds are loaded into their own buffers) should be played. There is only one modifier flag that can be used at the moment: DSBPLAY_LOOPING. This instructs DirectSound to play the sound in a loop. The absence of this flag means the sound will play once and then stop.

Now that we have the ability to play multiple sounds, we will need to write a new (CScene::CollectCallbacks) function that uses the sound manager to create and load these CSound objects and store them in callback keys in the actor's animation sets. Additionally, it would be nice if we could extract the sound data from another type of .ini file rather than hard coding the CollectCallbacks function to load the files we wish to play. In the next section we will discuss some upgrades to the actor's callback key collection system that will allow the callback key collection function (CScene::CollectCallbacks) access to the name of the file that contains the descriptions of the sound files we wish to load.

## The CollectCallbacks Callback Function

Before we discuss any changes, let us just refresh our memories with respect to the actor's callback key collection system.

As we know, when the LoadActorFromX method is called, it issues a call to the D3DXLoadMeshHierarchyfromX function to build the actor's hierarchy and its animation controller (if applicable). As we discussed in Chapter Ten, we are unable to register callback keys with an animation set that has already been created. Therefore, we implemented a private method of the actor called ApplyCallbacks which (when called from LoadActorFromX) would loop through every animation set that was loaded and call an application registered callback function (CScene::CollectCallbacks). This function would be passed an array of callback keys which it could fill with information. The application would register a special type of callback function with the actor prior to calling its LoadActorFromX function. This callback function would then be called for each loaded animation set. The callback function could create callback keys and pass them back to the actor. The actor would then clone the original animation set into a new one that has the callback keys registered with it.

You will remember that originally, the CScene::CollectCallbacks function loaded some wave files and stored them in CActionData objects. The pointers to these objects were then stored in the callback key itself as context data. When the callback was triggered, this pointer would be passed to the callback handling function which could then access the wave file stored inside and play it. You will also recall that the CActionData class was derived from IUnknown so that the actor could safely release objects of this type when the actor's controller was being destroyed.

This all worked fine, but when we have multiple sounds to load it can certainly become cumbersome to have to hard code the loading of each sound in the callback key's collection function. It would much nicer if we could store the sound files and the animation sets they should be registered for in an external file. That is exactly what we do in this lab project. You will find a file in the Data directory of this lab project called 'US Ranger.cbd' which contains the sound data that should be registered with the Walk_Legs_Only animation set. We will have a look at the contents of this file a little later on. For now, just know that it contains the names of sound files that should be registered with a particular animation set as callback keys.

> **Note:** The .cbd extension that we have given to our file is short for 'CallBack Data'. Although we are using this file to store sound data, it can be used to store any type of callback key data we wish to represent.

This raises an interesting question. If our CScene::CollectCallbacks function is going to create the callback information from a file, how does it know the name of that file, which may be different for different actors? The answer is simple -- we have to give the actor's COLLECTCALLBACKS callback function prototype an additional parameter so that other data can be passed by the actor to the callback function that creates the callback keys. This pointer can be used for anything, but in our code we use it to pass the name of the .cbd file containing the sound effect properties.

If we look in CActor.cpp we are reminded that the actor stores the pointer for a registered callback function of type CALLBACK_CALLBACKKEYS in the fourth element of its callback function array.

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

That is, if we wish to register a function 'SomeFunction' as a callback key collection function, we would do so like this:

```
pNewActor->RegisterCallback(CActor::CALLBACK_CALLBACKKEYS,somefunction,this );
```

As discussed in the past, the first parameter describes the type of callback we wish to register with the actor. In this example, it is a callback to create callback keys during actor creation. This will be called by the actor just after the animation data has been loaded, once for each animation set so the function has a chance to register keys with it. The second parameter is the function name itself, which in this example is called 'somefunction'. The third parameter is optional context information that will be passed through to the callback key creation function when it is called. In this example the 'this' pointer is being passed, which means the callback function will have access to member variables of the object that is registering the function. Remember, callback functions must be static if they are methods of a class.

Of course, the function signature for 'somefunction' must be defined in a very specific way as dictated by the actor. Previously, this function had to return a ULONG (the number of callback keys it placed into the passed array) and accept the four parameters shown below.

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

Every time this function is called by the actor, it will be passed the context that was passed into the RegisterCallback method (the *this* pointer for example), the name of the actor that is calling it, the name of the animation set that we are providing keys for, and a callback key array which we must fill with any callback keys we would like to register. This array is owned by the actor which calls this function just after the X file has been loaded.

When we look at the parameter list we can see that it provides no way for the name of the (.cbd) sound file to be passed. Therefore, we will modify the signature of this callback function (in CActor.h) to contain an additional parameter, as shown below.

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

As you can see, the actor now has the ability to pass an extra piece of information to the CollectCallbacks function in the form of a void pointer (pCallbackData). Our player's actor will use this to pass the name of a .cbd file to the callback function so that it knows where to load its sound data

from. The problem is, how does the actor know what data we want to send into this function? This too is now optionally passed into the LoadActorFromX method via a new parameter on the end.

```
HRESULT             LoadActorFromX  (     LPCTSTR FileName,
                                          ULONG Options,
                                          LPDIRECT3DDEVICE9 pD3DDevice,
                                          bool ApplyCustomSets = true,
                                          void * pCallbackData = NULL );
```

As you can see, when we call the actor's LoadActorFromX method we now have the option of passing in an additional void pointer. When the animation data has been loaded, the LoadActorFromX function will pass this pointer to the CActor::ApplyCallbacks method. This function originally took no parameters, but has been altered now so that it accepts a void pointer:

```
HRESULT ApplyCallbacks          ( void * pCallbackData );
```

As we know, this function loops through every loaded animation set in the actor, and if the CALLBACK_CALLBACKKEYS callback function has been registered, it will be called, passing in this new parameter as well as those shown above in the COLLECTCALLBACKS function prototype.

This means that the static CScene function that we register with the actor for callback key creation must also have this additional parameter added, so that it matches the layout defined for COLLECTCALLBACKS callback function. We will look at the actual code to the function later but below we see that its parameter list has been changed to match the correct signature for a callback key creation callback function.

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

## Representing our Sound Data

In Chapter Ten the CollectCallbacks function stored the names of the sound files we wanted to play in a class called CActionData. This was a very simple class that stored a string containing the name of the wave file. It was a pointer to an object of this type that was stored in the callback key so that it could be passed to the callback handler when the callback key was triggered. The callback key handler could then extract the sound file name and load and play it. This object also had to be derived from IUnknown and implement the AddRef, Release, and QueryInterface methods so that the actor could safely release these objects when the animation set was about to be destroyed. Although the actor knows nothing about the type of data we store in a callback key as context data, as long as it knows it is derived from IUnknown, before releasing the animation set it can fetch a pointer to each callback key's context data pointer, cast it to an IUnknown pointer and then call its Release method. This instructs the CActionData object (or

any IUnknown derived object) to destroy itself without the actor needing to have knowledge of what the object actually is and how its memory should be released.

The CActionData class served us well for previous demos, but now we will replace it with something a little more flexible. In CScene.h we declare an abstract base interface called IActionData to act as the base class from which any callback key context objects we register will be derived. This is a pure abstract class so it cannot be instantiated; it just sets out the methods that each callback key object must support.

The interface's GUID and definition is inside CScene.h, as shown below.

```
const GUID IID_IActionData    = {0xCCB7147E, 0x4480, 0x4C08, {0x8F, 0xC6, 0x95,
                                 0x57, 0xAC, 0x59, 0xA3, 0xA}};

class IActionData : public IUnknown
{
public:
      virtual HRESULT CreateCallback(     ULONG Index,
                                          LPVOID pContext,
                                          LPCTSTR strActorFile,
                                          void * pCallbackData,
                                          LPD3DXKEYFRAMEDANIMATIONSET pAnimSet,
                                          D3DXKEY_CALLBACK * pKey ) = 0;

      virtual HRESULT HandleCallback( UINT Track, CActor * pActor ) = 0;
};
```

As you can see, we now insist that not only should a callback object support the methods from IUnknown for lifetime encapsulation purposes, but it must now implement two methods called CreateCallback and HandleCallback. In order to understand the parameter lists to these functions, let us look at what the .cbd file looks like that we created for Lab Project 12.3.

***Excerpt from US Ranger.cbd***

```
[Walk_Legs_Only]

CallbackCount               = 4

; First Callback (Left foot touches ground)
Callback[0].Ticks           = 20
Callback[0].Action          = PLAY_SOUND
Callback[0].PlayMode        = RANDOM
Callback[0].SampleCount     = 4
Callback[0].Sample[0].File  = Data\Sounds\Footsteps L1.wav
Callback[0].Sample[1].File  = Data\Sounds\Footsteps L2.wav
Callback[0].Sample[2].File  = Data\Sounds\Footsteps L3.wav
Callback[0].Sample[3].File  = Data\Sounds\Footsteps L4.wav

; Second Callback (Right foot touches ground)
Callback[1].Ticks           = 40
Callback[1].Action          = PLAY_SOUND
Callback[1].PlayMode        = RANDOM
Callback[1].SampleCount     = 4
```

```
Callback[1].Sample[0].File  = Data\Sounds\Footsteps R1.wav
Callback[1].Sample[1].File  = Data\Sounds\Footsteps R2.wav
Callback[1].Sample[2].File  = Data\Sounds\Footsteps R3.wav
Callback[1].Sample[3].File  = Data\Sounds\Footsteps R4.wav

; First Callback (Left foot touches ground)
Callback[2].Ticks           = 60
Callback[2].Action          = PLAY_SOUND
Callback[2].PlayMode        = RANDOM
Callback[2].SampleCount     = 4
Callback[2].Sample[0].File  = Data\Sounds\Footsteps L1.wav
Callback[2].Sample[1].File  = Data\Sounds\Footsteps L2.wav
Callback[2].Sample[2].File  = Data\Sounds\Footsteps L3.wav
Callback[2].Sample[3].File  = Data\Sounds\Footsteps L4.wav

; Second Callback (Right foot touches ground)
Callback[3].Ticks           = 80
Callback[3].Action          = PLAY_SOUND
Callback[3].PlayMode        = RANDOM
Callback[3].SampleCount     = 4
Callback[3].Sample[0].File  = Data\Sounds\Footsteps R1.wav
Callback[3].Sample[1].File  = Data\Sounds\Footsteps R2.wav
Callback[3].Sample[2].File  = Data\Sounds\Footsteps R3.wav
Callback[3].Sample[3].File  = Data\Sounds\Footsteps R4.wav
```

The layout of this file is very simple, especially in this case where we are only defining the information for one animation set.

At the top of the file we can see that the only block defined is for the animation set called ‘Walk_Legs_Only’. This entire file is just defining a number of callbacks for the one set. If we had callback key data to define for multiple animation sets, there would be multiple sections in this file arranged one after another.

Inside the Walk_Legs_Only section of the file we specify the callback key data. The first property we must set is called CallbackCount, which specifies the number of callback keys we would like to register with this animation set. The ‘Walk_Legs_Only’ animation set is a short looping walking sequence which walks the legs in a ‘Left-Right-Left-Right’ motion. That is, each foot makes contact with the ground twice. Since we wish to trigger a sound whenever a foot touches the ground, this means we will need to register four callback keys to play four footstep sounds at different times.

Following the CallbackCount property is the actual callback key definitions list. Each property should use the format Callback[*N*].*Value* where *N* is the index of the callback key that we are defining for that particular animation set. That is, the indices are local to the section. *Value* should be replaced with the name of the property we are defining.

Each callback key should have at least two properties in common regardless of what type it is. It should have a Ticks property which describes where in the periodic timeline of the animation set this key should be triggered and an Action property which describes what type of callback key definition it is. Currently, we have only defined one type of callback key definition called PLAY_SOUND, but you can define you own custom callback key types and develop the objects to parse and handle them.

A PLAY_SOUND callback definition encapsulates the registration of a sound event with an animation set. However, we can assign multiple sound effects to this callback key. For example, for each of our callback keys we have defined four wave files to be played. This allows us to either cycle through this sound list each time the callback key is triggered or we can allow our sound playing object to choose one from the list at random. You will notice that for our first callback key definition we specify the names of four wave files. These represent four slightly different sound effects of a foot hitting the ground. We also set the play mode to RANDOM so that we know we can just choose any wave file from the list when this callback key is triggered. You will notice that each of the four keyframes is set up the same way. Having multiple sound effects for a single key really helps in this case. If we had just a single footstep sound it would be very repetitive and would produce a 'Thud Thud Thud Thud' sound as the player walks. Because each key when triggered will be choosing a footstep sound to play at random we get a non-repetitive footsteps sound like 'Thud Crunch Stomp Thud Stomp Crunch Crunch'. The end result is much more pleasing because it sounds like the player is walking over an uneven terrain with many different materials underfoot. The other option for this value is CYCLE, which means we wish to step through the list in order every time the callback key is triggered (with wraparound back to the top of the list when all sounds have been played).

As you can see, if the callback key definition is of type PLAY_SOUND then the PlayMode, SampleCount, and SampleList properties are also expected to be defined. If you wish to register a single sample then just set sample count to 1 and list only one wave file for that key.

Now that we know how our data is laid out in the .cbd file, let us have a look at the methods of the IActionData interface and discuss there parameter lists.

## The IActionData Interface

```
virtual HRESULT CreateCallback(          ULONG Index,
                                         LPVOID pContext,
                                         LPCTSTR strActorFile,
                                         void * pCallbackData,
                                         LPD3DXKEYFRAMEDANIMATIONSET pAnimSet,
                                         D3DXKEY_CALLBACK * pKey ) = 0;
```

The IActionData interface is the base class for objects which encapsulate callback key actions (e.g., playing a sound). The CreateCallback function will be called to extract the information for a given key as each IActionData derived object encapsulates the data for a single callback key. That is, it is an object of this type that will have its pointer stored along with the callback key in the animation set and will contain the information that must be triggered when the callback is executed.

The CreateCallback method will be called by the CScene::CollectCallbacks method whenever it wishes to create a new callback key. It will create a new IActionData derived object and call this method for a given animation set, passing in the index of the callback key. This index is used by our derived object to know which callback key we should extract the information from the .cbd file for. If Index was set to 2, the IActionData object should extract and encapsulate the information for the second callback key for the animation set passed (as the fifth parameter). As the second parameter, context data can also be

passed. This is always a pointer to the CScene object in our demo applications. As the third parameter the name of the actor is passed and as the fourth parameter the name of the .cbd file is passed (in our example, but this can be whatever context data was passed into the LoadActorFromX function). As the fifth parameter we pass in the animation set whose callback key we wish to create. The final parameter is a pointer to a D3DXKEY_CALLBACK structure that will eventually store a pointer to this IActionData derived object before being registered with the set.

```
virtual HRESULT HandleCallback( UINT Track, CActor * pActor ) = 0;
```

The HandleCallback function is not called during callback registration (inside CScene::CollectCallbacks); it is called inside the callback handler function when the callback key is triggered. The callback handler will be passed a pointer to the IActionData derived object. The object is expected to know what it has to do. For example, if our object is a sound player object, then it should know that when its HandleCallback function is called, it should play that sound without having to burden the CScene object with the responsibility (as we did previously). We will see this function being used later when we look at the code to the callback handler.

In our project we derive a class called CSoundCallback from IActionData. It is used to encapsulate a single callback key's sound data. In our example, that means we will create four callback keys for our Walk_Legs_Only animation set and each will store the address of a CSoundCallback object as its context pointer. We will look at the code to this object in a moment, but for now just know that its CreateCallback method will extract and load the sound data from the .cbd file for the given callback key that it represents and its HandleCallback function will know how to pick a random sound from the list and play it.

At this point, we can get some clarity by looking at the new implementation of the CScene::CollectCallbacks function. Remember, this function is called from an actor just after its animation data has been loaded. It will be called once for each animation set, allowing this function a chance to add some callback keys to it. The final parameter to this function is a D3DXKEY_CALLBACK array that was allocated by the actor. You simply store any callback keys you would like to register in this array and on function return, the actor will clone the animation set and add the callback keys you specified to the set.

```
ULONG CScene::CollectCallbacks(  LPVOID pContext,
                                 LPCTSTR strActorFile,
                                 void * pCallbackData,
                                 LPD3DXKEYFRAMEDANIMATIONSET pAnimSet,
                                 D3DXKEY_CALLBACK pKeys[] )
{
    IActionData * pData  = NULL;
    LPTSTR strDefinition = (LPTSTR)pCallbackData;
    LPCTSTR strSetName   = pAnimSet->GetName();
    ULONG CallbackCount, i, KeyCount = 0;
    TCHAR strBuffer[128], strKeyName[128];

    // Retrieve the number of callbacks defined
    CallbackCount = GetPrivateProfileInt( strSetName, _T("CallbackCount"),
                                          0,
                                          strDefinition );
```

```
    if ( CallbackCount == 0 ) return 0;
```

The first thing this function does is cast the pCallbackData to a string as this will contain the name of the .cbd file that was passed by the application into the CActor::LoadActorFromX function. It is stored in the strDefinition local variable. It then fetches the name of the passed animation set so that it can be determined whether this function is currently being called for a set we have any interest in registering callback keys for. We then try and retrieve the CallbackCount property for the section in the file that matches the name of the current animation set we are processing. If there is no definition for this animation set then zero will be returned by default and we can exit.

At this point, if CallbackCount is non-zero it means there was a section defined in the file for the current animation set we are processing.

In the next section of code we loop through the number of callback key definitions contained in the .cbd file for the current animation set we are processing. For each one, we first grab the value of its Ticks property. Since this is a float, we have to read it in as string and then use sscanf to convert. Notice how we store the extracted periodic position (Ticks) in the Time member of the current element in the passed D3DXKEY_CALLBACK array. KeyCount is a local variable that was initialized to zero at the top of the function and is incremented every time we fill in a new structure in this array.

```
    // Loop through each of the callbacks specified
    for ( i = 0; i < CallbackCount; ++i )
    {
        // Retrieve the 'time' for this particular callback
        _stprintf( strKeyName, _T("Callback[%i].Ticks"), i );

        GetPrivateProfileString( strSetName,
                                 strKeyName, _T("0.0"),
                                 strBuffer,
                                 127,
                                 strDefinition );

        // Scan directly into key time
        sscanf( strBuffer, _T("%g"), &pKeys[ KeyCount ].Time );
```

The next bit is interesting. We extract the Action property of the current callback key definition which allows us to determine what type of callback key this represents. As we currently only support keys of type PLAY_SOUND, if it is not of the correct type we skip it. However, if it is a PLAY_SOUND key definition, we create a new CSoundCallback object.

```
        // Retrieve the 'action' type for this particular callback
        _stprintf( strKeyName, _T("Callback[%i].Action"), i );
        GetPrivateProfileString(    strSetName,
                                    strKeyName,
                                    _T(""),
                                    strBuffer,
                                    127,
                                    strDefinition );

        // What type of callback is this
```

```
        if ( _tcsicmp( strBuffer, _T("PLAY_SOUND") ) == 0 )
            pData = new CSoundCallback( GetGameApp()->GetSoundManager() );
        else
            continue;

        // Validate
        if ( !pData ) continue;
```

We have not looked at the code to the CSoundCallback class yet, but this is our object derived from IActionData that will encapsulate the loading and playing of sound keys. Note that as the sole parameter to its constructor we pass in a pointer to the application's CSoundManager. The CSoundCallback class will need access to the sound manager so that it can create and play CSound objects.

At this point, we have created the new CSoundCallback object for the current key, so we next call its CreateCallback function. This function will extract all the sound information from the .cbd file for the current key and create CSound objects for each sound. They will be stored in a global array of sounds that can be played for this key. Notice that we pass in loop variable 'i' so that it knows the index of the key it needs to extract the data for. If 'i' was set to 2, it would know that it must extract the properties of the second key definition in the file for the current animation set being processed. Also note that the final parameter is the address of the callback key element in the passed array where this object will eventually have its pointer stored. We do not use this parameter in our CSoundCallback::CreateCallback function, but it is useful to have access to it for any custom objects you might develop later.

```
        // Allow the callback to initialize itself
        if (FAILED(pData->CreateCallback( i,
                                          pContext,
                                          strActorFile,
                                          pCallbackData,
                                          pAnimSet,
                                          &pKeys[ KeyCount ] ) ) )
        {
            delete pData;
            continue;

        } // End if failed to create callback
```

At this point the CSoundCallback object will have been created for the current callback key being processed. It will contain an array of CSound objects which can be played at random when the key is triggered.

All that is left to do at the bottom of the loop is store a pointer to this CSoundCallback object in the callback key array and increase the callback key count. When this callback key is triggered, the callback handler will be passed a pointer to this CSoundCallback object. The handler can then issue a call to the CSoundCallback::HandleCallback function which will instruct the object to select a CSound object at random from its internal array and play it.

```
        // Store in the key's data area
        pKeys[KeyCount].pCallbackData = (LPVOID)pData;

        // We've successfully added a key
```

```
        KeyCount++;

    } // Next Callback

    // Return the total number of keys created
    return KeyCount;
}
```

We have seen how simple the CScene::CollectCallbacks function is. Most of the heavy lifting has been placed inside the CSoundCallback object so let us have a look at the code to this object next.

## The CSoundCallback Class

The CSoundCallback class encapsulates all the data for a single PLAY_SOUND callback key. It is defined in CScene.h as shown below. We have snipped the list of functions from the listing to compact listing size. We already know the five methods it exposes: AddRef, Release, and QueryInterface inherited from IUnknown and CreateCallback and HandleCallback inherited from IActionData. Here we see the list of its member variables and structures followed by a description of each.

***Excerpt from CScene.h***

```
class CSoundCallback : public IActionData
{
public:

    enum PlayStyle { PLAY_LOOPING = 0, PLAY_ONCE = 1 };

    … Snip… (Functions)

private:
    struct SampleItem
    {
        TCHAR   FileName[512];          // Filename for the loaded sample
        CSound *pSound;                 // The actual sound object.
    };

    ULONG           m_nRefCount;        // Reference counter
    bool            m_bRandom;          // We should pick a random sample
    ULONG           m_nSampleCount;     // Number of samples stored
    ULONG           m_nNextPlayIndex;   // Which sample did we last play?
    SampleItem    * m_pSamples;         // The array which stores all the samples loaded.
    CSoundManager * m_pSoundManager;    // The sound manager specified by our application
};
```

**ULONG m_nRefCount**

This variable holds the current reference count of the object. Our CSoundCallback object must implement the same reference counting mechanism as a COM object so that the actor can safely release this object using the IUnknown::Release method. This is important because, although a pointer to this object will be stored in the callback key, it will be stored as a void pointer and the actor has no idea what sort of object it is or how to delete it. As long as the actor can rely on the fact that the object stored in each callback key is derived from IUnknown, it can cast the void pointer to an IUnknown pointer and instruct the object to destroy itself.

**bool m_bRandom**
This boolean will be set to true if this CSoundCallback object has been placed into random playlist mode. When in random mode, every time this object's HandleCallback method is called it will randomly select a sample to play from its internal array. We saw when we examined the contents of the .cbd file that each of our four keys were set to random mode. Therefore, the four CSoundCallback objects that we create will each have this variable set to true.

**ULONG m_nSampleCount**
This contains the number of sampled sounds this object stores in its internal sound list. It describes the number of SampleItem structures in the array discussed below. For each of the CSoundCallback objects we create in our demo this value will be set to 4 because each callback key definition in the .cbd file has four wave file sounds associated with it.

**SampleItem * m_pSamples**
This is an array of SampleItem structures. Each element in the array contains the name of the sound file that is stored there and a pointer to the CSound object that was created for that sound.

```
struct SampleItem
{
    TCHAR   FileName[512];    // Filename for the loaded sample
    CSound *pSound;           // The actual sound object.
};
```

In a moment we will take a look at the CSoundCallback::CreateCallback function, which is called just after the object is instantiated in CScene::CollectCallbacks (shown earlier). This function extracts the wave file names from the passed .cbd file and uses the application's CSoundManager object (passed in the constructor) to create a CSound object for each wave file listed. These pointers are then stored in this array along with the original name of the sound file (taken from the .cbd file). It is this list that we choose a sound effect to play from each time the key is triggered and the object's HandleCallback function is called. We will see this array having its contents created when we look at the CreateCallback method in a moment.

**ULONG m_nNextPlayIndex**
This value is used only if the object is placed into CYCLE mode. It stores the index of the current position in the sound list so we know which sound to play next when the HandleCallback method is called. This value will start of at zero and be incremented every time the HandleCallback method is called and a sound is played. When this value reaches the end of the list it will be wrapped back around to zero so we start playing sounds from the start of the list again.

**CSoundManager *m_pSoundManager**
In this member we store a pointer to the CSoundManager object which will be used to create CSound objects. This pointer must be passed into the constructor as we saw earlier when the object was being created in the CScene::CollectCallbacks function (we passed in the CGameApp's CSoundManager object pointer).

Let us now look at the methods of the CSoundCallback object, which are all very simple.

## CSoundCallback::CSoundCallback

We already saw the constructor being called from the CScene::CollectCallbacks function. It is passed the application's CSoundManager pointer which it stores in a member variable. It also switches the object's playlist into CYCLE mode (random=false) by default and sets the sample array pointer to NULL. We also set the initial reference count of the object to 1.

```
CSoundCallback::CSoundCallback( CSoundManager * pSoundManager )
{
    // Setup default values
    m_pSoundManager     = pSoundManager;
    m_bRandom           = false;
    m_pSamples          = NULL;
    m_nSampleCount      = 0;
    m_nNextPlayIndex    = 0;

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

}
```

Notice how we also set the play index to zero so that the first sound to be played in cycling mode will be the first one loaded into the sample array.

## CSoundCallback::~CSoundCallback

The destructor is also very simple. Its main task is to release its array of SampleItem structures if it exists. Keep in mind that each structure in this array will store a pointer to a CSound object, so we must release that too.

```
CSoundCallback::~CSoundCallback( )
{
    ULONG i;

    // Release any memory
    if ( m_pSamples )
    {
        for ( i = 0; i < m_nSampleCount; ++i )
        {
            // Delete the sound object if it exists.
            if ( m_pSamples[i].pSound ) delete m_pSamples[i].pSound;
        } // Next Sample

        delete []m_pSamples;

    } // End if samples loaded

    // Clear variables
    m_bRandom           = false;
```

```
    m_pSamples          = NULL;
    m_nSampleCount      = 0;
    m_nNextPlayIndex    = 0;
}
```

## CSoundCallback::CreateCallback

This is the function where all of our object initialization happens. It is called from the CScene::CollectCallbacks function when it is determined that there is a callback key definition in the .cbd file for the current animation set being processed. When this function is called it is passed the index of the callback key inside the .cbd file that this object is to encapsulate. For example, if the Index parameter passed was 4 and the fifth parameter was a pointer to an animation set with the name 'Walk_Legs_Only', that would mean this object is to fetch the data for the fourth callback key in the .cbd section called [Walk_Legs_Only]. The function is also passed the callback function context data. This is an arbitrary data pointer that is registered with the actor at the same time the callback function itself is registered. We always store the CScene pointer along with the callback function so that the static CScene::CollectCallbacks function can access the non-static members of the object instance. The function is also passed a string containing the name of the actor that is having the callback keys registered and an additional context data pointer called pCallbackData. This is the pointer we send into the LoadActorFromX function which (in our application) contains the name of the .cbd file that we want the CollectCallbacks function to use. A pointer to the interface of the animation set currently requesting callback key registration is also passed. The final parameter is a pointer to the D3DXKEY_CALLBACK structure that will eventually store a pointer to this CSoundCallback object.

This function must extract the data from the .cbd file and create a new CSound object for each sound file listed for this key. These CSound objects are then stored in the SampleItem array so that they can be played when the key is triggered by the animation system. Here is the first section of code.

```
HRESULT CSoundCallback::CreateCallback(   ULONG Index,
                                          LPVOID pContext,
                                          LPCTSTR strActorFile,
                                          void * pCallbackData,
                                          LPD3DXKEYFRAMEDANIMATIONSET pAnimSet,
                                          D3DXKEY_CALLBACK * pKey )
{
    LPTSTR strDefinition = (LPTSTR)pCallbackData;
    LPCTSTR strSetName   = pAnimSet->GetName();
    TCHAR strBuffer[128], strKeyName[128];
    ULONG i;

    // Random or cycle sample playlist?
    _stprintf( strKeyName, _T("Callback[%i].PlayMode"), Index );

    GetPrivateProfileString(  strSetName,
                              strKeyName,
                              _T("CYCLE"),
                              strBuffer,
                              127,
                              strDefinition );
```

```
    if ( _tcsicmp( strBuffer, _T("RANDOM") ) == 0 ) m_bRandom = true;
```

Notice that we store the name of the .cbd file (pCallbackData) in the strDefinition local variable and store the name of the animation set in the strSetName local variable. We then fetch the play mode for the current key. The name of the key is built by substituting the Index parameter into the string "Callback[Index].Playmode". Remember, Index is the local index of the key we are fetching for a given animation set. We then use the GetPrivateProfileString method to fetch this property. The parameters specify that we want to retrieve a property from a section with a name that matches the name of the animation set, with a key name of 'Callback[*n*].PlayMode' where *n* is the current index of the callback key. If the property is not found, a default string of "CYCLE" will be returned, placing the object into non-random mode. As the final parameter we pass in strDefinition, which now contains the name of the .cbd file we wish to extract the data from.

When the function returns, the strBuffer array should contain the value of this property (either RANDOM or CYCLE). If it contains the word "RANDOM", then the m_bRandom boolean property of the object is set to true, placing the object into random playlist mode.

In the next section we fetch the sample count of the current key and allocate the object's array of SampleItem structures to hold that many samples.

```
    // Retrieve the number of samples we're using
    _stprintf( strKeyName, _T("Callback[%i].SampleCount"), Index );

    m_nSampleCount=GetPrivateProfileInt(strSetName, strKeyName, 0,strDefinition );
    if ( m_nSampleCount == 0 ) return E_INVALIDARG;

    // Allocate enough space for the sample data
    m_pSamples = new SampleItem[ m_nSampleCount ];
    if ( !m_pSamples ) return E_OUTOFMEMORY;

    // Clear out array.
    ZeroMemory( m_pSamples, m_nSampleCount * sizeof( SampleItem ) );
```

Now we will set up a loop that will fetch the name of each sample from the file and will store it in the SampleItem array. We will then pass this filename to the CSoundManager::Create method so that it can create a CSound object for the respective sound file. Notice how the m_pSamples[i].pSound pointer is passed as the first parameter to the Create method so that on function return, the new CSound object created will have its pointer stored in the correct place in the sample array.

```
    // Loop through and retrieve the items
    for ( i = 0; i < m_nSampleCount; ++i )
    {
        // Retrieve sample filename
        _stprintf( strKeyName, _T("Callback[%i].Sample[%i].File"), Index, i );

       GetPrivateProfileString(     strSetName,
                                    strKeyName,
                                    _T(""),
                                    m_pSamples[i].FileName,
```

```
                                    511,
                                    strDefinition );

        // Attempt to load the sound
        m_pSoundManager->Create( &m_pSamples[i].pSound, m_pSamples[i].FileName );

    } // Next Sample

    // Success!
    return S_OK;
}
```

So far we have covered all the steps involved in storing our CSoundCallback pointers in the callback keys for a given animation set. In summary:

1. The COLLECTKEYS callback function is registered with actor prior to load call.
2. Call to LoadActorFromX is passed a context pointer that will be passed to the callback function registered in step 1.
3. After the actor has loaded the animation data it will loop through each loaded animation set and call the callback function, passing it the context data pointer passed to the actor in step 2. We use this to pass a .cbd filename to the function (in our lab project).
4. The callback function is called by the actor to provide any callback keys for the current animation set being processed. This function opens the .cbd file to see if there is a section specific to the current animation for which the callback function in being invoked. If a section exists with the same name as the animation set, it knows there must be callback keys to set up.
5. For each key definition for a given animation set, a CSoundCallback object is created and the CSoundCallback::Create method is called to build its internal playlist of CSound objects.
6. The CSoundCallback object pointer, along with the Ticks value of that key (extracted from the cbd file), is stored in a D3DXKEY_CALLBACK structure and placed in the callback key array that is returned to the actor.
7. The actor will clone the animation set to create a new one with the desired keys.

At this point then, our actor will be fully created and each animation set that we desire will have their respective callback keys defined. The callback keys themselves store a pointer to a CSoundCallback object which is passed to the callback handler function when the key is triggered. The callback handler function can then simply call the CSoundCallback::HandleCallback method which will instruct it to pick and play a new sound. Let us have a look at the HandleCallback method next.

## CSoundCallback::HandleCallback

This function has a very simple job. It will be called by the callback handling function every time the callback key is triggered. It has to decide which sound to play from its play list (based on its mode of operation) and play that sound.

The Handle Callback function accepts two parameters, although this particular object's implementation of the function uses neither of them. However, the parameter list laid out in the base interface dictates that this function should be passed the index of the track on the controller from which the callback key is

being triggered and a pointer to the actor who is currently using that controller. Although this function is very simple and has no need for these parameters, you can certainly imagine how your own callback key objects (derived from IActionData) may well need them. For example, you might use the track index to fetch the weight of the track that triggered the key. Perhaps a sound is only played if the weight is high enough. We might imagine for example that we want to trigger a sound for the arms when moving up into the gun position (a gun reload sound perhaps). However, the shooting pose is a single frame set and as such, we can not just register a sound with it as it will repeat for as long as the character is holding his gun to his shoulder. One way around this would be to fetch the weight of the track and only trigger the reload sound if the weight was between say 0.5 and 0.6. This would mean his arms are roughly 50% of the way through the transition into the shooting pose. Although this is not a great example, it demonstrates why having the track index can be useful. Of course, having the track by itself is not useful at all if we have no idea which animation controller this track index is related to. Therefore, a pointer to an actor is passed so that the function can retrieve its controller's track properties.

In our case the function is simple -- we just wish to play a sound. If the m_bRandom boolean variable is set to true then we generate a random index into the SampleItem array. If not, then we are in cycle mode and the m_nNextPlayIndex variable will hold the index of the next sound to play (0 initially). We use our index to access the SampleItem array and call the Play method of the CSound object that is stored there. Here is all of the code:

```
HRESULT CSoundCallback::HandleCallback( UINT Track, CActor * pActor )
{
    ULONG PlayIndex;

    // Select item from 'play list'
    if ( m_bRandom )
        PlayIndex = rand() % m_nSampleCount;
    else
        PlayIndex = m_nNextPlayIndex;

    // Play the sample if it's valid
    if ( m_pSamples[PlayIndex].pSound ) m_pSamples[PlayIndex].pSound->Play( );

    // Cycle the next play index
    m_nNextPlayIndex++;
    m_nNextPlayIndex %= m_nSampleCount;

    // Success!
    return S_OK;
}
```

Notice that after the sound has been played, the m_nNextPlayIndex variable is incremented and wrapped around to zero if it exceeds the maximum sample count.

That covers the code for the CSoundCallback object. This class also has AddRef, Release, and QueryInterface methods, but you should be very comfortable with these types of functions by now, so will not show or discuss their implementation again here.

## Updating CActionHandler

In Chapter Ten we learned how to use the callback handler features of the D3DXAnimationController. In short, we had to derive a class from the ID3DXAnimationCallbackHandler abstract base class. This base interface specified a single method that must be implemented in the derived class, called HandleCallback. We pass an instance of this object to the AdvanceTime method and the HandleCallback method will be invoked whenever a callback key is triggered. Nothing much has changed in this project except that now our CActionHandler class has an added CActor member pointer and an extra method to set it. Here is our updated CActionHandler object:

**Except from CScene.h**

```
class CActionHandler : public ID3DXAnimationCallbackHandler
{
public:
    CActionHandler( );

    STDMETHOD(HandleCallback)(THIS_ UINT Track, LPVOID pCallbackData);


    // CActionHandler
    void SetCurrentActor( CActor * pActor );

private:
    CActor * m_pActor;  // Storage for the actor that is currently being advanced
};
```

Why have we added the actor pointer? As we discussed earlier, when a callback key is triggered, the CActionHandler::HandleCallback method will be called by the controller and passed the index of the track that triggered the event. We are also passed a void pointer to the individual key's context data (which in our case will be a pointer to the CSoundCallback object for that key). This function can then simply call the CSoundCallback object's HandleCallback method because, as we have seen, it is this method which takes care of actually playing the sound. Below we show our implementation of the CActionHandler::HandleCallback function. Remember, it is this method that is called by the D3DXAnimationController when its parent object is passed into the AdvanceTime call. If we take a look at the HandleCallback method of this object it will become clear why that actor pointer is needed.

### CActionHandler::HandleCallback

Below we see all of the code to the handling function that is called by the controller when a callback key is triggered. It simply casts the passed data pointer to an IUnknown pointer and then uses its QueryInterface method to make sure it is a pointer to an IActionData derived object. Once we have an IActionData pointer, we can call its HandleCallback method. In the case of our CSoundCallback object, it will choose a sound to play from its list of sounds. Notice however that the IActionData::HandleCallback function expects to be passed a pointer to an actor; information that is not provided by the animation controller. We need the current actor being updated stored in the handler

object so that the track ID we are passed actually means something. Without us knowing what controller spawned the callback, the track index is useless to us.

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

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

    // This is our baby, lets handle it
    pAction->HandleCallback( Track, m_pActor );

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

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

So, we need to place an additional function call in our CScene::AnimateObjects function. Prior to calling AdvanceTime and passing in the handler object, it will call the handler object's SetCurrentActor method. Below we show the CScene::AnimateObjects method with this additional function call highlighted in bold.

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

    // Process each object for coll det
    for ( i = 0; i < m_nObjectCount; ++i )
    {
        CObject * pObject = m_pObject[i];
        if ( !pObject ) continue;

        // Get the actor pointer
        CActor * pActor = pObject->m_pActor;
        if ( !pActor ) continue;
        if ( pObject->m_pAnimController )
             pActor->AttachController( pObject->m_pAnimController,
                                       false,
                                       pObject->m_pActionStatus );

        // Set this actor in the handler so the callbacks have access to it
        Handler.SetCurrentActor( pActor );

        // Advance time
        pActor->AdvanceTime( Timer.GetTimeElapsed(), false, &Handler );

    } // Next Object
}
```

As you can see, we create a handler object on the stack at the top of the function and pass the same object into the AdvanceTime call for each actor. Therefore, before we call AdvanceTime, we set the handler's actor pointer to the current actor we are processing so that when the callback handler is called by the controller, it has access to the actor from which it was triggered. The callback function then has the ability to query the actor to find out the state of its controller and the track index passed in becomes useful.

That concludes our coverage of the updated callback key system. Hopefully you will find it relatively straightforward to use. In the absence of a proper application sound manager, it provides us a fair bit of flexibility for such a small amount of effort.

# Application Framework Upgrades

We are approximately halfway through the course at this point. Moving forward we will spend the remainder of the course discussing topics less related to the DirectX API and more related to general games programming. So before we do that, we will use Lab Project 12.2 to make some changes to existing classes to make our framework more usable in future lessons. We will also upgrade our framework's IWF code to provide more comprehensive support for the various entity types exported from GILES™. This will allow us to consolidate many of the topics and techniques we have learned about so far in the entire series. Our application framework will soon support heightmap based terrains, actors, and static scenes, all within our CScene class. We will also add support for the loading of fog parameters from IWF files and the rendering of the scene using those fog parameters. From here on in, our application framework should be able to load and render most IWF files exported from GILES™ that we care to throw at it, even if that IWF file contains multiple terrain entities, external references to animated X files, and static geometry.

## A Modular Approach to Querying Supported Texture Formats

In previous applications our CGameApp class would query the support of texture formats that its various components would need. The querying was done in CGameApp::TestDeviceCaps which was called at application startup from the CGameApp::InitInstance method. For example, if the terrain class needed an alpha texture format for its water plane, the test for a suitable alpha format was performed by CGameApp and the resulting format was passed to the terrain object for storage. It could then create textures using this format when the terrain was being initialized. Likewise, if the scene required support for compressed textures, the CGameApp::TestDeviceCaps function would have to be modified to test for a suitable compressed texture format, which would then be passed to the scene object for storage and texture creation during scene initialization. This approach served us fairly well when we are only plugging a few components into our framework, but as we start plugging in more components which may each have different texture needs, it becomes cumbersome.

Rather than having to alter the code to our CGameApp class every time we plug in a new component that requires a texture format we have not previously provided support for, it would be much nicer if the component itself could intelligently chose which format it wants to use to complete its task. However,

we do not want each component to have to duplicate the device tests that are done in the CGameApp::TestDeviceCaps method.

The solution we provide in this lab project, which will be used in all future projects, introduces a new class called CTextureFormatEnum. This class contains a list of all texture formats that are supported by DirectX paired with a boolean value indicating whether or not this texture format is supported by the current device.

The CGameApp object will contain a member of this type and in the TestDeviceCaps method it will now simply call the CTextureFormatEnum::Enumerate method. This method will loop through every DirectX supported format and test for device compliance. When this function returns, the CTextureFormatEnum will contain a 'true' or 'false' property paired with each possible texture format.

There is nothing too clever about this, but what this new class also has are methods that allow any object to say things like "Give me the best compressed texture format with an alpha channel". The object will have logic that will loop though each supported format and return the best one that matches the criteria.

Rather than each component that we plug into framework (CScene, CTerrain etc) having to be passed supported texture formats by the CGameApp class, the CGameApp object will essentially just call this object's Enumerate method to allow us to find which formats are supported. Each component we plug in to our framework can then simply be passed a pointer to this CTextureFormatEnum object so that the component can query its method to retrieve the formats it desires. The CGameApp class no longer has to be continually edited and extended to support more formats every time we plug in a new component with different texture needs. Instead, we just make sure that CGameApp calls the SetTextureFormat method for the component and passes in a pointer to this texture format enumeration object. We will have to make sure that all the components (e.g., CTerrain, CScene, etc.) have a function of this type and the ability to store a pointer to the enumeration object.

If you look at the CGameApp class, it now has a new member as shown below:

**Excerpt from CGameApp.h**

```
CTextureFormatEnum      m_TextureFormats;
```

This new member is an instance of the type of the new class we are about to create. Looking inside the CGameApp::TestDeviceCaps method we can see that it no longer performs any texture format queries on the device; instead it simply calls the Enumerate method of this new object.

**Excerpt From CGameApp::TestDeviceCaps**

```
ULONG      Ordinal = pSettings->AdapterOrdinal;
D3DDEVTYPE Type    = pSettings->DeviceType;
D3DFORMAT  AFormat = pSettings->DisplayMode.Format;

// Enumerate texture formats the easy way :)
if ( !m_TextureFormats.Enumerate( m_pD3D, Ordinal, Type, AFormat )) return false;
```

We will cover all the code to the CTextureFormatEnum object in a moment. For now, just know that by calling its Enumerate method and passing in the current device, the adapter ordinal, the type of device

(HAL for example) and the current adapter format, the object will have its internal data filled with a list of supported formats.

At this point, any component with a pointer to this object can use its methods to ask for supported formats. The following code snippet is taken from the CGameApp::BuildObjects function and shows the pointer to this object being passed to the CScene object for storage.

```
m_Scene.SetTextureFormat( m_TextureFormats );
m_Scene.LoadSceneFromIWF( m_strLastFile, LightLimit )
```

Notice how we call the SetTextureFormat method before calling the loading function. This allows the scene to use this object to query for supported texture formats during the loading process. What is also nice is that the scene object can pass this pointer to any of its components (e.g., CTerrain objects) so that they too can use the object to query texture formats independently.

Because the scene object now has a pointer to the CTextureFormatEnum object, whenever it needs to create a texture, it can just call the object's GetBestFormat method to return the most desirable format for the current device.

Below we see a line of code from the CScene::CollectTexture function, which we use to load all the scene's textures. In this example, the GetBestFormat method is being used with no parameters. This means it will use the default logic when choosing the best format -- finding the best non-alpha compressed texture format. There will also be a version of this function that allows you to override this logic by specifying request parameters

**Excerpt from CScene::CollectTexture**

```
D3DXCreateTextureFromFileEx( pScene->m_pD3DDevice,
  Buffer,
  D3DX_DEFAULT, D3DX_DEFAULT,
  3, 0,
  pScene->m_TextureFormats.GetBestFormat(),
  D3DPOOL_MANAGED,
  D3DX_DEFAULT,
  D3DX_DEFAULT,
                              0,
  NULL,
  NULL,
  &pNewTexture->Texture );
```

Notice how easy it is for a component to fetch the best compatible format now.

## The CTextureFormatEnum Object

The declaration and code for this class are stored in the files 'CTextureFormatEnum.h' and 'CTextureFormatEnum.cpp' respectively. Below we see the class declaration minus its methods to compact the listing. We will discuss its methods and the code they contain in a moment.

**Excerpt from CTextureFormatEnum.h**

```
typedef std::map<D3DFORMAT, bool> FormatEnumMap;

class CTextureFormatEnum
{
public:
    …
    …
    // Methods not Shown here
    …
    …
    FormatEnumMap   m_TextureFormats;       // Formats
    bool            m_bPreferCompressed;    // Compressed Texture Bias Mode
    bool            m_bAlpha;               // Should we query alpha formats only?
};
```

The class has three member variables. The last two are simply booleans that contain the default mode of the query methods (discussed in a moment). That is, if m_bAlpha is set to true then when we ask the object to return the best supported format, it will prefer one with alpha if it exists and return that. Likewise, if the m_bPreferCompressed boolean is set to true, the query methods will favor compressed formats. The states of these booleans can be set and retrieved via simple member functions. They can also be overridden by passing parameters into the query functions. Having these defaults just provides a nice way for the application to set the most common parameters that will be required so that it can call the parameterless version of GetBestFormat when it wants to fetch a compatible texture format.

The top-most member is of type FormatEnumMap. This is a typedef for an STL map object that stores a D3DFORMAT and a boolean in key/value pairs. For those of you a little rusty with maps, a map is a way for us to essentially build a table of key-value pairs (like a dictionary). We might imagine that an array is like a linear map where the keys are the array indices and the values are stored in those elements. However, in a map, the keys can be anything you like (floats, strings, or even entire structures). Internally, maps are stored as binary trees, which makes the searching for a key/value pair very quick indeed.

So why are we using a map? We are using map because each key will be a D3DFORMAT structure and the value associated with that key will be a Boolean that is set to true or false depending on whether the format is supported. This allows our object, for example, to record support for the X8R8G8B8 texture format as shown below.

```
m_TextureFormats[ D3DFMT_X8R8G8B8 ] = true;
```

With that in mind, let us examine the code to our new CTextureFormatEnum object.

## CTextureFormatEnum::CTextureFormatEnum

The constructor is simple. The default behavior is to prefer compressed formats over non-compressed ones and to prefer non-alpha formats unless an alpha format is explicitly requested when the query is performed. In the final step, the constructor calls the Init method of the object which initializes the STL map of texture formats so that nothing is initially supported until the Enumerate method is called.

```
CTextureFormatEnum::CTextureFormatEnum()
{
    // Setup any required values
    m_bPreferCompressed = true;
    m_bAlpha            = false;

    // Initialize the class
    Init();
}
```

## CTextureFormatEnum::Init

This function first empties the STL map by calling its Clear method. At this point there will be no key/value pairs in the map. We then add the keys for each possible texture format. We use a default value of false since we have not yet queried any of these formats on the device to test for support.

```
void CTextureFormatEnum::Init()
{
    // First clear the internal map if it's already been used.
    m_TextureFormats.clear();

    // Setup each of the map elements
    m_TextureFormats[ D3DFMT_R8G8B8        ] = false;
    m_TextureFormats[ D3DFMT_A8R8G8B8      ] = false;
    m_TextureFormats[ D3DFMT_X8R8G8B8      ] = false;
    m_TextureFormats[ D3DFMT_R5G6B5        ] = false;
    m_TextureFormats[ D3DFMT_X1R5G5B5      ] = false;
    m_TextureFormats[ D3DFMT_A1R5G5B5      ] = false;
    m_TextureFormats[ D3DFMT_A4R4G4B4      ] = false;
    m_TextureFormats[ D3DFMT_R3G3B2        ] = false;
    m_TextureFormats[ D3DFMT_A8            ] = false;
    m_TextureFormats[ D3DFMT_A8R3G3B2      ] = false;
    m_TextureFormats[ D3DFMT_X4R4G4B4      ] = false;
    m_TextureFormats[ D3DFMT_A2B10G10R10   ] = false;
    m_TextureFormats[ D3DFMT_A8B8G8R8      ] = false;
    m_TextureFormats[ D3DFMT_X8B8G8R8      ] = false;
    m_TextureFormats[ D3DFMT_G16R16        ] = false;
    m_TextureFormats[ D3DFMT_A2R10G10B10   ] = false;
    m_TextureFormats[ D3DFMT_A16B16G16R16  ] = false;
    m_TextureFormats[ D3DFMT_A8P8          ] = false;
    m_TextureFormats[ D3DFMT_P8            ] = false;
    m_TextureFormats[ D3DFMT_L8            ] = false;
    m_TextureFormats[ D3DFMT_A8L8          ] = false;
    m_TextureFormats[ D3DFMT_A4L4          ] = false;
```

```
    m_TextureFormats[ D3DFMT_V8U8           ] = false;
    m_TextureFormats[ D3DFMT_L6V5U5         ] = false;
    m_TextureFormats[ D3DFMT_X8L8V8U8       ] = false;
    m_TextureFormats[ D3DFMT_Q8W8V8U8       ] = false;
    m_TextureFormats[ D3DFMT_V16U16         ] = false;
    m_TextureFormats[ D3DFMT_A2W10V10U10    ] = false;
    m_TextureFormats[ D3DFMT_UYVY           ] = false;
    m_TextureFormats[ D3DFMT_R8G8_B8G8      ] = false;
    m_TextureFormats[ D3DFMT_YUY2           ] = false;
    m_TextureFormats[ D3DFMT_G8R8_G8B8      ] = false;
    m_TextureFormats[ D3DFMT_DXT1           ] = false;
    m_TextureFormats[ D3DFMT_DXT2           ] = false;
    m_TextureFormats[ D3DFMT_DXT3           ] = false;
    m_TextureFormats[ D3DFMT_DXT4           ] = false;
    m_TextureFormats[ D3DFMT_DXT5           ] = false;
    m_TextureFormats[ D3DFMT_L16            ] = false;
    m_TextureFormats[ D3DFMT_Q16W16V16U16   ] = false;
    m_TextureFormats[ D3DFMT_MULTI2_ARGB8   ] = false;
    m_TextureFormats[ D3DFMT_R16F           ] = false;
    m_TextureFormats[ D3DFMT_G16R16F        ] = false;
    m_TextureFormats[ D3DFMT_A16B16G16R16F  ] = false;
    m_TextureFormats[ D3DFMT_R32F           ] = false;
    m_TextureFormats[ D3DFMT_G32R32F        ] = false;
    m_TextureFormats[ D3DFMT_A32B32G32R32F  ] = false;
    m_TextureFormats[ D3DFMT_CxV8U8         ] = false;

}
```

This function really shows us what the object stores -- a big list of possible texture formats with a boolean describing its support status on the device. Of course, this is all changed when the Enumerate method is executed (usually the first method our application will call).

## CTextureFormatEnum::Enumerate

This function is called to update the status of each key in the STL map to reflect the support offered by the device. The function is passed a pointer to the device and the adapter ordinal. It is also passed the type of device we are interested in enumerating formats for (HAL for example) and the current adapter format since this is a factor in determining what texture formats are compatible.

The first thing we do is set up an iterator to begin searching through the key/values pairs in the map. Each key/value pair of a map is accessed by the map iterator object using its ‘first’ and ‘second’ members. That is, as the iterator accesses each row of map data, its ‘first’ member will contain the key (the format itself, such as D3DFMT_X8R8G8B8) and the ‘second’ member can be used to set or retrieve the value associated with that key (the boolean status).

Once we setup the iterator to point at the first row of map data, we will loop through all the rows added in the previous function. We will then use the IDirect3DDevice9::CheckDeviceFormat function to query support for each mode and set the boolean status accordingly. At the end of this function, any modes that are supported in the map will have a ‘true’ value associated with them.

Here is the code. First we use the map's Begin method so the iterator is pointing to the first row of data. We then set up a loop that increments the iterator until the End is reached. We call the CheckDeviceFormat function for the current key and set its value to true if the function succeeds.

```
bool CTextureFormatEnum::Enumerate( LPDIRECT3D9 pD3D,
                                    ULONG Ordinal,
                                    D3DDEVTYPE DeviceType,
                                    D3DFORMAT AdapterFormat )
{
    FormatEnumMap::iterator Iterator = m_TextureFormats.begin();

    // Validate
    if ( !pD3D ) return false;

    // Loop through all texture formats in the list
    for ( ; Iterator != m_TextureFormats.end(); ++Iterator )
    {
        // Test the format
        if ( SUCCEEDED( pD3D->CheckDeviceFormat( Ordinal,
                                                 DeviceType,
                                                 AdapterFormat,
                                                 0,
                                                 D3DRTYPE_TEXTURE,
                                                 Iterator->first ) ) )
        {
            // It's supported, mark it as such
            Iterator->second = true;

        } // End if supported

    } // Next Texture Format Item

    // Success!
    return true;
}
```

## CTextureFormatEnum::GetBestFormat

This function is called to retrieve the best supported format. This function (it is overloaded) accepts two parameters that allow the caller to override the default logic of the process. The two booleans are used to specify whether we consider a format with an alpha channel to be a priority and whether compressed formats are preferred over regular formats. This function returns a D3DFORMAT which can then be used to create texture surfaces on the current device.

The first section of the function shown below is the conditional code block that is executed when a compressed format is preferred.

```
D3DFORMAT CTextureFormatEnum::GetBestFormat( bool Alpha, bool PreferCompressed )
{
    // Does the user prefer compressed textures in this case?
    if ( PreferCompressed )
```

```
    {
        // Alpha is required?
        if (Alpha)
        {
            if ( m_TextureFormats[D3DFMT_DXT3] ) return D3DFMT_DXT3;
            if ( m_TextureFormats[D3DFMT_DXT5] ) return D3DFMT_DXT5;

        } // End if alpha required
        else
        {
            if ( m_TextureFormats[D3DFMT_DXT1] ) return D3DFMT_DXT1;

        } // End if no alpha required

    } // End if prefer compressed formats
```

As you can see above, if alpha is preferred we return D3DFMT_DXT3 as the most desirable format if its boolean is set to true in the map. This is a desirable compressed alpha surface because it provides several levels of alpha and it stores the alpha information explicitly. That is to say, each pixel within the texture maintains its own alpha value. If this format is not supported, we return the next best compressed alpha format: D3DFMT_DXT5. This format still provides several levels of alpha, but much of the original alpha data is lost in the compression. Alpha information is interpolated over a wider area of pixels.

**Note:** There was a detailed discussion of compressed texture formats in Chapter 6 of Module I.

Finally, if alpha is not required then we return D3DFMT_DXT1. This is a compressed format with no support for multiple levels of alpha. This surface only reserves a single bit for alpha information (meaning the pixel is either on or off). This is a most desirable compressed texture format if several levels of alpha are not required since it allows for greater compression (because of the missing alpha information).

That concludes the compressed texture search. The next code block is executed if alpha is required and either non-compressed textures were preferred or a suitable compressed texture could not be found in the above code. In the latter case, even if compressed alpha textures were desired, we will fall back to finding non-compressed alpha formats if the function did not return in the compressed code block.

Our logic is simple -- we are searching for the texture format with the most color and alpha resolution. For alpha requests, we are ideally looking for a 32-bit surface with 8 bits of storage for the red, green, blue, and alpha components. If this is not available, we will make a choice that might seem odd -- we will test support for the A1R5G5B5 format. While this only supports one bit of alpha, as compared to the A4R4G4B4 format which has more bits for alpha and may also be supported, in this latter mode we also only have 4 bits per RGB component. This lack of color resolution generally looks pretty bad. In all situations then, we always give higher priority to the RGB components, even if it means we are returning a texture with only one bit (on/off) of alpha. As the third alpha option we fall back to the A4R4G4B4 format

```
    // Standard formats, and fallback for compression unsupported case
    if ( Alpha )
    {
        if ( m_TextureFormats[ D3DFMT_A8R8G8B8 ] ) return D3DFMT_A8R8G8B8;
        if ( m_TextureFormats[ D3DFMT_A1R5G5B5 ] ) return D3DFMT_A1R5G5B5;
        if ( m_TextureFormats[ D3DFMT_A4R4G4B4 ] ) return D3DFMT_A4R4G4B4;

    } // End if alpha required
```

The next section of code is executed if the caller requested either a non-compressed, non-alpha format, or if a compressed non-alpha format was requested but a solution was not found in the compressed code block. Once again, you can see that we favor resolution for the RGB components in the order that we test and return supported non alpha surfaces.

```
    else
    {
         if ( m_TextureFormats[ D3DFMT_X8R8G8B8 ] ) return D3DFMT_X8R8G8B8;
         if ( m_TextureFormats[ D3DFMT_R8G8B8   ] ) return D3DFMT_R8G8B8;
         if ( m_TextureFormats[ D3DFMT_R5G6B5   ] ) return D3DFMT_R5G6B5;
         if ( m_TextureFormats[ D3DFMT_X1R5G5B5 ] ) return D3DFMT_X1R5G5B5;

    } // End if no alpha required

    // Unsupported format.
    return D3DFMT_UNKNOWN;
}
```

If the bottom line of the function is ever reached, it means we could not find a format that was supported (very unlikely) and we return D3DFMT_UNKNOWN back to the caller. The caller will then know that his request could not be satisfied.

## CTextureFormatEnum::GetBestFormat (Overloaded)

The GetBestFormat function is also overloaded so that we can call it with no parameters and use the values of its m_bPreferCompressed and m_bAlpha member variables to control the searching logic. This function simply calls the version of the function we just covered passing in the member variables as the parameters.

This was the version of the function we saw being called in the earlier example of its use.

```
D3DFORMAT CTextureFormatEnum::GetBestFormat( )
{
    return GetBestFormat( m_bAlpha, m_bPreferCompressed );
}
```

Of course, we can also set the m_bAlpha and m_bPreferCompressed booleans via the two simple methods shown below, so that the default logic of the object can be changed.

```
void CTextureFormatEnum::SetCompressedMode( bool Enable )
{
    m_bPreferCompressed = Enable;
}
```

```
void CTextureFormatEnum::SetAlphaMode( bool Enable )
{
    m_bAlpha = Enable;
}
```

## CTextureFormatEnum::GetFormatSupported (Overloaded)

There may be times when you do not wish to use the default processing supplied by the GetBestFormat functions, but instead wish to query if a given format is supported. Your application can then use our object simply to test support for surfaces and make up its own mind about which one is preferred.

The GetFormatSupported function takes a single parameter of type D3DFORMAT that describes the surface you would like to query support for. The function then uses the STL map Find method to search the map for a matching key. Of course, the Enumerate method must have been called first for any of these query methods to work.

If the key exists in the map, then the iterator will be pointing at that key/value pair when the Find method returns. If not, the iterator will be pointing at the end of the map data and we know that we do not have a row in our map for the specified format. We have added all the common formats to the map, so this should never happen in the normal case (not unless you are using a custom format).

```
bool CTextureFormatEnum::GetFormatSupported( D3DFORMAT Format ) const
{
    FormatEnumMap::const_iterator Item = m_TextureFormats.find( Format );

    // We don't support the storage of this format.
    if ( Item == m_TextureFormats.end() ) return false;

    // Return the item
    return Item->second;
}
```

As you can see, we simply return the value of the iterator's 'second' member, which will be pointing to the boolean value for the key (the format) that was specified. Therefore, this function returns true or false to indicate support for that format.

## CTextureFormatEnum::SetFormatSupported

The final function we will cover in this class is the SetFormatSupported function. The application can use this to force the boolean value associated with a format to be true or false. As the first parameter we pass the format (the key) whose value we wish to set. As the second parameter we pass a boolean that indicates the support (or lack of it) for the specified format.

```
void CTextureFormatEnum::SetFormatSupported( D3DFORMAT Format, bool Supported )
{
    FormatEnumMap::iterator Item = m_TextureFormats.find( Format );

    // We don't support the storage of this format.
    if ( Item == m_TextureFormats.end() ) return;

    // Update the item (done this way because we have already had to perform
    // a 'find'. It saves us a little time over using the indexing operator
    // to find the key again).
    Item->second = Supported;
}
```

We now have a texture format enumeration object that all our future components can use. All we have to do is make sure that any component we plug into our framework can be passed and store a pointer to the enumeration object. From that point on, the component can use this object to query for formats that it is interested in using.

## Multiple Terrains in CScene

Since Chapter Three in Module I, our demonstration applications have generally taken one of two forms. They have either been terrain demos or non-terrain demos. The non-terrain demos used the CScene class as a geometry manager and loaded data from IWF files or X files. We have continued to use and expand the CScene class throughout Module II.

In previous terrain oriented demos, because the generation of the terrain geometry happened procedurally (based on heightmap data), a different scene manager was used, called CTerrain. The CTerrain class essentially replaced CScene in those demos. The data was no longer loaded from an IWF file and as such, lab projects that contained terrains used different code than those that did not. Although using the CTerrain class as a substitute scene manager for terrain based applications served us well in earlier lessons to keep the code simple and focused, this system will have to be revised going forward.

In a game we often do not want our scene to consist of either indoor or outdoor environments; often we want a scene to be comprised of both. Our current system does not support this idea, so we will upgrade the CScene and CTerrain classes so that they can be used together in a single application.

The CScene class will now officially be the top level scene manager and it will continue to load its data from IWF files. However, we will add support for the loading of terrain entities from the IWF file so that the IWF file can contain internal geometry and external references in addition to the terrain. In fact, a scene that we make in GILES™ may contain multiple terrain entities, so our scene will have to create, manage, and render multiple CTerrain objects.

> **Note**: You will see later that when a terrain exists as an entity in an IWF file, the geometry is not stored. All that is stored is the information used to construct it -- the heightmap file, the filenames of the base and detail textures, the scale of the detail texture and the scale of the geometry. These are all variables that were member variables in our old CTerrain class and were used to construct the terrain. Therefore, all we have to do is write a function that allows us to send these parameters into a terrain object when it is first created and we can use virtually the same code as before, with the exception of one or two little adjustments that will be discussed in a moment.

It may at first seem that changing the terrain class and the scene class to work harmoniously with one another would require many code changes. In the grand scheme of things, it requires hardly any. We will first discuss changes to the CScene class.

Just as the CScene class maintains an array of objects, meshes, and actors that collectively compose the virtual world, the CScene class will now also maintain an array of CTerrain objects. The following lines of code have been added to the class declaration of CScene:

**Excerpt from CScene.h**

```
CTerrain          **m_pTerrain;
ULONG               m_nTerrainCount;
```

As you can see, the scene now manages an array of CTerrain pointers and has an added ULONG describing how many CTerrain pointers are currently in the array. Every time the loading code encounters a terrain entity, it will create a new CTerrain object from the data extracted from the IWF file and add its pointer to this array.

## CScene::AddTerrain

Just as the CScene class has an AddObject, an AddMesh, and an AddActor function to resize the respective arrays and make room for another element at the end, we now have to have an AddTerrain method which does the same for terrains.

This function is called by the IWF loading code whenever a terrain entity is encountered in the file. The code to this function should be easily understood since it is almost identical to all the other array resize functions we have written in previous lessons.

AddTerrain takes a single optional parameter that allows the caller to specify how many new elements should be added to the end when the array resize is performed. The default value of the parameter is 1, so this function will make space in the terrain array for one additional CTerrain pointer.

If previous terrain pointer data exists in the array it is copied over into the new resized array before the original array is deleted. The new resized array is then assigned to the CScene::m_pTerrain pointer replacing the original assignment to the old array.

```
long CScene::AddTerrain( ULONG Count /* = 1 */ )
{
    CTerrain ** pTerrainBuffer = NULL;

    // Allocate new resized array
    if (!( pTerrainBuffer = new CTerrain*[ m_nTerrainCount + Count ] )) return -1;

    // Existing Data?
    if ( m_pTerrain )
    {
        // Copy old data into new buffer
        memcpy( pTerrainBuffer, m_pTerrain, m_nTerrainCount * sizeof(CTerrain*) );

        // Release old buffer
```

```
        delete []m_pTerrain;

    } // End if

    // Store pointer for new buffer
    m_pTerrain = pTerrainBuffer;

    // Clear the new items
    ZeroMemory( &m_pTerrain[m_nTerrainCount], Count * sizeof(CTerrain*) );

    // Increase Terrain Count
    m_nTerrainCount += Count;

    // Return first Terrain
    return m_nTerrainCount - Count;
}
```

## CScene::Render

As you are well aware, our scene is rendered when the CGameApp::FrameAdvance function calls the CScene::Render function. It is this function that instructs the various components comprising the scene to render themselves. It first instructs the skybox to render itself, and then it loops through every object in the scene instructing that object's mesh or actor to render itself.

We will now have to add an additional line to the CScene::Render function that instructs any terrains that may be used by the scene to render themselves as well. We will not cover the entire CScene::Render function at this time, as it has hardly changed at all. However, below we see a new line of code from that function which renders each terrain in the CScene::m_pTerrain array. This line of code is positioned very near the bottom of the function, just after the loop that renders each CObject.

```
// Render each terrain
for ( i = 0; i < m_nTerrainCount; ++i ) m_pTerrain[i]->Render( &Camera );
```

With the exception of the CScene IWF loading code upgrade which we will cover at the end of this workbook, these are the only changes that need to be added to the CScene class so that it now correctly renders multiple terrains. Next let us discuss the updates we will need to apply to our CTerrain class to get everything working together.

## The New CTerrain Class

There is no need for concern since we do not have to write a new terrain class. With a few minor modifications, our old CTerrain class can be upgraded to work alongside our new CScene, and alongside other CTerrain objects that might exist in the scene.

> **Note**: This object was first covered in Chapter Three of Module I when we explained how the vertex data for the terrain was generated from a heightmap. We also described how the terrain was actually represented internally as a series of CMesh objects. A CMesh object was really just a lite wrapper around a vertex and index buffer with functions that allowed us to add vertices and indices incrementally to the object during the building process. As this object is no longer used anywhere else other than by the CTerrain class, we have decided to rename CMesh to CTerrainBlock. This is just a simple name change and all the code remains the same, unless stated otherwise. If you are feeling a little rusty on how we generated our terrain, please refer back to Chapter Three. You may also want to refresh yourself on the workbook of Module I Chapter Six where we discussed the concept of texturing the terrain with a base and detail texture map. All of this code will remain the same, so will not be covered again here.

The only real problem we need to address is that in our old CTerrain class the vertex data was assumed to be generated in world space. That is, the terrain space origin was also the world space origin. As the data was assumed to be in world space, rendering the terrain simply meant setting the world matrix to an identity matrix and sending the vertex and index data of each terrain block to the hardware.

In an IWF file, the terrain data is defined in model space and is accompanied by a world matrix describing where in the scene this terrain should be positioned when rendered or queried for collision. Therefore, we will have to add a world matrix member to our CTerrain class. Before the terrain renders itself (in the CTerrain::Render function) it will need to send this world matrix to the device. This simple addition means we can now load an IWF file that contains multiple terrain entities that are all placed at different locations in the world. We can thus position and render these terrains correctly.

Our CTerrain class will also have a method added called SetTextureFormat so that it can be passed (by CGameApp or CScene) a pointer to the CTextureFormatEnum object introduced earlier. The terrain creation functions can then use this object to find the optimal texture format to use when loading the base and detail texture map images.

Furthermore, we will upgrade our CTerrain object to provide lifetime encapsulation using COM-like semantics. That is, we will add the AddRef and Release mechanics so that the CTerrain object can destroy itself when its internal reference count reaches zero.

Finally, the LoadTerrain method will be written to replace the old LoadHeightMap method as the means for generating the terrain geometry. Previously, the application would call the CTerrain::LoadHeightMap method and pass the name of the .RAW file that contains the height data and the width and the height of that image. This function would then generate the terrain based on the heightmap data and the values of several internal variables (e.g., geometry scale, texture scale, etc.). If we wanted to change the generation properties of the terrain, we would have to alter the code and re-compile.

The new LoadTerrain method accepts a single parameter -- a pointer to a TerrainEntity structure. Not only does this single structure contain all the terrain creation parameters we used before, it also matches the format in which the data for a terrain is stored in an IWF file. Therefore, when our loading code encounters a terrain entity, it can just pull the data from the file into a TerrainEntity structure and pass it straight to the CTerrain::LoadTerrain method for geometry generation. This is very handy indeed.

Below we show the new methods added to CTerrain in the file 'CTerrain.h'. Not all of the class methods are show here in order to keep the listing compact, but we have included the ones that are new and those that have been modified. We have also shown the member variables and highlighted new or modified ones in bold face type.

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


    // Public Functions For This Class
    …
    …
    …
    void                SetTextureFormat ( const CTextureFormatEnum &FormatEnum );
    void                SetWorldMatrix   ( const D3DXMATRIX & mtxWorld );

    bool                LoadTerrain      ( TerrainEntity * pTerrain );
    const D3DXMATRIX &  GetWorldMatrix   ( ) const {return m_mtxWorld; }
    …
    …
    …

    // Reference Counting Functions
    ULONG               AddRef              ( );
    ULONG               Release             ( );


    // Public Static Functions For This Class (callbacks)
    static void UpdatePlayer(LPVOID pContext, CPlayer *pPlayer, float TimeScale );
    static void UpdateCamera(LPVOID pContext, CCamera *pCamera, float TimeScale );

private:

    // Private Variables For This Class

    D3DXVECTOR3         m_vecScale;         // Amount to scale the terrain meshes
    D3DXVECTOR2         m_vecDetailScale;   // Amount to scale the detail map.
    float              *m_pHeightMap;       // The physical heightmap data loaded
    ULONG               m_nHeightMapWidth;  // Width of the 2D heightmap data
    ULONG               m_nHeightMapHeight; // Height of the 2D heightmap data

    CTerrainBlock     **m_pTerrainBlocks;   // Simple array of mesh pointers
```

```
    ULONG               m_nBlockCount;      // Number of blocks stored here
    LPDIRECT3DDEVICE9   m_pD3DDevice;       // D3D Device
    bool                m_bHardwareTnL;     // Used hardware vertex processing ?
    bool                m_bSinglePass;      // Use single pass rendering method

    CTextureFormatEnum  m_TextureFormats;   // Texture format lookup object

    LPTSTR              m_strDataPath;      // Path to data items
    ULONG               m_nPrimitiveCount;  // Number of primitives for D3D Render
    LPDIRECT3DTEXTURE9  m_pBaseTexture;     // Base terrain texture
    LPDIRECT3DTEXTURE9  m_pDetailTexture;   // Terrain detail texture.

    D3DXMATRIX          m_mtxWorld;         // Terrain world matrix
    ULONG               m_nRefCount;        // Reference Count Variable

    // Private Functions For This Class
    long                AddTerrainBlock     ( ULONG Count = 1 );
    bool                BuildTerrainBlocks  ( );
    void                FilterHeightMap     ( );
};
```

There are a couple things to pay attention to in the above listing. The terrain now has a world matrix and also has Get/Set functions that allow for the configuration of this matrix by external components. It also has a reference count variable with accompanying AddRef/Release methods. So the terrain now behaves like a COM object and destroys itself when there are no outstanding references to it. Finally, notice that what was once a CMesh pointer array containing the terrain blocks is now an array of CTerrainBlocks. This is just an object renaming for clarity and no code has changed. Thus the function AddMesh has had its name changed to AddTerrainBlock and the function BuildMeshes has had its name changed to BuildTerrainBlocks.

## The TerrainEntity Structure

The TerrainEntity structure is defined in CTerrain.h and is our property transport structure when informing the CTerrain object how its geometry should be built. It also describes exactly how the terrain data exported from GILES™ into the IWF file is laid out in that file.

We will see later when we cover the upgraded loading code that for each terrain entity found in the file we will create a new CTerrain object and pass a structure of this type to its LoadTerrain function. Of course, you can fill one of these structures out manually if you wish to generate your terrain creation properties from another source and not use GILES™ or IWF files. The members of this structure should be instantly familiar to you since they are the properties that existed in our previous CTerrain class member variables. We will briefly discuss the members as a reminder.

```
typedef struct _TerrainEntity
{
    ULONG       TerrainType;        // Which type of terrain are we using.
    char        TerrainDef[256];    // Either terrain definition (splats) or
                                       heightmap def (standard).
```

```
    char        TerrainTex[256];    // (Standard terrain only), specifies the base
                                        texture
    char        TerrainDet[256];    // (Standard terrain only), specifies the
                                        detail texture
    ULONG       TerrainWidth;       // (Standard terrain only), Width of the
                                        heightmap
    ULONG       TerrainHeight;      // (Standard terrain only), Height of the
                                        heightmap
    D3DXVECTOR3 TerrainScale;       // (Standard terrain only), Scale of the
                                        terrain
    D3DXVECTOR2 TerrainDetScale;    // (Standard terrain only),
                                    //  scale information for detail texture
    ULONG       Flags;              // Reserved Flags

} TerrainEntity;
```

**ULONG TerrainType**
The GILES™ terrain entity can be one of two types -- a regular terrain or a splatting terrain. You will recall that the data is generated rather differently for each terrain type. Our CTerrain object encapsulates regular heightmapped terrains. This value will be stored in the entity to inform the loading code which type of terrain entity has been encountered. A value of zero indicates a regular terrain entity. Therefore, our CScene class will only be interested in loading regular terrain entities, as that is the type of terrain this object currently encapsulates. Thus, this value should always be zero when this structure is used to create CTerrain objects (at least for now).

**char TerrainDef[256]**
This member contains the filename of the heightmap image that the CTerrain object will need to load and extract the height values from. For splatting terrains, this will contain the name of the splat definition file that contains all the splat terrain info.
(The following members are applicable only to regular terrains)

**char TerrainTex[256]**
This member contains the filename of the image file that should be loaded into a texture and used as the base texture for the terrain.

**char TerrainDet[256]**
This member contains the filename of the image file that should be loaded into a texture and used to tile over the terrain as a detail texture.

**ULONG Width**
**ULONG Height**
As the image data for the terrain heightmap is stored in a .RAW file that contains no data describing the arrangements of the pixels in the image, these members inform us how the data in the .RAW file is to be interpreted in terms of rows and columns. For example, if Width=1024 and Height=512, the pixels in the .RAW file should be used to create a mesh of 512 rows of 1024 vertices.

**D3DXVECTOR3 TerrainScale**
This vector describes how the pixel positions in the heightmap are scaled into terrain space vertex positions. For example, a scale vector of (1,1,1) would produce a terrain where every unit in world space maps to every pixel in image space and the height data for any pixel will be used exactly as the world space Y component of that vertex. A scale vector of (1,5,2) on the other hand, would scale the pixel position by 1 along the X axis, 2 along the Z axis, and the Y component of the vertex would be generated by scaling the value stored in the pixel by 5. Using this second example scale vector, a raw file containing 10 rows and 10 columns would create a terrain 10 units wide (X axis) and 20 units deep (Z axis) and at its peak would be five times higher than the highest point described by the height data.

**D3DXVECTOR2 TerrainDetScale**
This 2D vector describes how many times we would like the detail texture tiled over the terrain in the direction of the X and Z axes.

## CTerrain::LoadTerrain

The LoadTerrain method is used to instruct a newly created CTerrain object to build its geometry and load its texture resources. This function is almost identical to the LoadHeightMap method in our old CTerrain class with the exception that it now extracts the properties for creation from the passed TerrainEntity structure.

We will show the code to this function below but provide only very brief descriptions since it should be familiar to you. We will not show the code to any of the helper functions it calls since these were all covered in Module I and are unchanged. Please refer back to Chapters Three and Six if you need to jog your memory about how these functions work. This function is shown as a reminder of the terrain creation process from a high level.

The first thing this function does is return if the TerrainType member of the passed TerrainEntity structure is not zero. This means it is not a regular terrain and as such, not a terrain this class was designed to support. We also return if the device is not valid or if its array of terrain blocks has already been allocated. If data already exists in the terrain, it should be released prior to calling this function. We then extract the width and height values from the passed structure and store them in member variables.

```
bool CTerrain::LoadTerrain( TerrainEntity * pTerrain )
{
    HRESULT   hRet;
    FILE    * pFile = NULL;
    TCHAR     FileName[MAX_PATH];
    D3DFORMAT fmtTexture;
    ULONG     i;

    // Skip if this is not a 'standard' terrain
    if ( !pTerrain || pTerrain->TerrainType != 0 ) return false;

    // Cannot load if already allocated (must be explicitly released for reuse)
    if ( m_pTerrainBlocks ) return false;

    // Must have an already set D3D Device
```

```
    if ( !m_pD3DDevice ) return false;

    // First of all store the information passed
    m_nHeightMapWidth  = pTerrain->TerrainWidth;
    m_nHeightMapHeight = pTerrain->TerrainHeight;
```

In the next section we combine the name of the highway file with the string containing our application's data path so that we have the complete path of the RAW file we wish to load. We then open the file and read in its total size, so that we know how much data we will need to read.

```
    // Build full filename
    _tcscpy( FileName, (m_strDataPath) ? m_strDataPath : pTerrain->TerrainDef );
    if ( m_strDataPath ) _tcscat( FileName, pTerrain->TerrainDef );

    // Open up the heightmap file
    pFile = _tfopen( FileName, _T("rb") );
    if (!pFile) return false;

    // Get file length
    ULONG FileSize = filelength( pFile->_file );
```

We next calculate how many rows and columns our heightmap must have using the passed width and height values in the TerrainEntity structure. For example, if FileSize contains the number of pixels in the image, dividing this by the height of the heightmap will tell us the length of a single row. The number of rows is then calculated by dividing the file size by the length of a row. We also perform some logic to calculate the height and width if only one of them has been specified in the terrain entity structure (and the other property is set to zero). If the TerrainEntity structure has zero in both its width and height members, then we will assume the image file is to be interpreted as a square heightmap. In that case the width and height will be equal to the square root of the total file size.

```
    // Work out heightmap size if possible
    if ( m_nHeightMapWidth > 0 && m_nHeightMapHeight == 0 )
    {
        // We can work this one out
        m_nHeightMapHeight = FileSize / m_nHeightMapWidth;

    } // End if m_nHeightMapWidth only
    else if ( m_nHeightMapWidth == 0 && m_nHeightMapHeight > 0 )
    {
        // We can work this one out
        m_nHeightMapWidth = FileSize / m_nHeightMapHeight;

    } // End if m_nHeightMapHeight only
    else
    {
        // We can only assume square
        m_nHeightMapWidth  = (ULONG)sqrt( (double)FileSize );
        m_nHeightMapHeight = m_nHeightMapWidth;

    } // End if no sizes at all
```

In the next step we test that the file size is large enough to have enough data for the width and height values we have just calculated. If the file size is smaller than 'Width x Height' then something has gone wrong and we return. We also retrieve the terrain geometry scale and the detail texture scale vectors and copy them into member variables.

```
    // Validate sizes
    if ( m_nHeightMapWidth * m_nHeightMapHeight > FileSize )
        { fclose( pFile); return false; }

    // Retrieve scale
    m_vecScale = pTerrain->TerrainScale;
    m_vecDetailScale = pTerrain->TerrainDetScale;
```

Now we allocate an array of floats large enough to store a floating point value for each pixel in the heightmap. This is the array we will copy the height values into for each vertex of the terrain we are about to create. We then set up a loop to read the value of each pixel and store it in the heightmap array before we close the file.

```
    // Attempt to allocate space for this heightmap information
    m_pHeightMap = new float[m_nHeightMapWidth * m_nHeightMapHeight];
    if (!m_pHeightMap) return false;

    // Read the heightmap data
    for ( i = 0; i < m_nHeightMapWidth * m_nHeightMapHeight; i++ )
    {
        UCHAR HeightValue;
        fread( &HeightValue, 1, 1, pFile );

        // Store it as floating point
        m_pHeightMap[i] = (float)HeightValue;

    } // Next Value

    // Finish up
    fclose( pFile );
```

We now call the FilterHeightMap method which applies a smoothing filter to the height data so that our terrain does not look jagged because of the integer height values stored in the file. That is why we read the integer height data into a float array -- once we have the height data stored as floats we can apply a smoothing filter to soften the peaks and valleys. The code to this function was covered in Module I and is unchanged.

```
    // Filter the heightmap data
    FilterHeightMap();
```

Our next step is to load the base texture and the detail texture. Before calling the D3DXCreateTextureFromFileEx function to load these textures, we call the GetBestFormat method of our CTextureFormatEnum object. This will return a D3DFORMAT type that we can pass to the loading functions for texture generation. Before we load each texture, we append the filename to the

application's data path so we have the full path and filename of the image file to pass to the texture loading function.

```
    // Get the best format for these textures
    fmtTexture = m_TextureFormats.GetBestFormat( );

    // Load in the textures used for rendering the terrain
    _tcscpy( FileName, (m_strDataPath) ? m_strDataPath : pTerrain->TerrainTex );
    if ( m_strDataPath ) _tcscat( FileName, pTerrain->TerrainTex );
    hRet = D3DXCreateTextureFromFileEx( m_pD3DDevice,
                                        FileName,
                                        D3DX_DEFAULT,
                                        D3DX_DEFAULT,
                                        D3DX_DEFAULT,
                                        0,
                                        fmtTexture,
                                        D3DPOOL_MANAGED,
                                        D3DX_DEFAULT,
                                        D3DX_DEFAULT,
                                        0, NULL, NULL, &m_pBaseTexture );


    _tcscpy( FileName, (m_strDataPath) ? m_strDataPath : pTerrain->TerrainDet );
    if ( m_strDataPath ) _tcscat( FileName, pTerrain->TerrainDet );
    hRet = D3DXCreateTextureFromFileEx(   m_pD3DDevice,
                                          FileName,
                                          D3DX_DEFAULT,
                                          D3DX_DEFAULT,
                                          D3DX_DEFAULT,
                                          0,
                                          fmtTexture,
                                          D3DPOOL_MANAGED,
                                          D3DX_DEFAULT, D3DX_DEFAULT,
                                          0,
                                          NULL,
                                          NULL,
                                          &m_pDetailTexture );
```

At this point the heightmap data is loaded and filtered and the texture assets have been loaded. All that is left to do is generate the terrain blocks.

We first call the AddTerrainBlock method to allocate enough space in the terrain block pointer array for each terrain block we will create. The number of terrain blocks the heightmap data is divided into depends on how many quads we wish to place in each block. The QuadsWide and QuadsHigh variables can be altered and are also defined in CTerrain.h. By default, they are set to 16 so that each terrain block contains a 16x16 block of quads.

```
    // Allocate enough terrain blocks to store the separate blocks of this terrain
    if ( AddTerrainBlock( ((m_nHeightMapWidth - 1) / QuadsWide) *
                          ((m_nHeightMapHeight - 1)/ QuadsHigh) ) < 0 )
       return false;
```

The AddTerrainBlock function is not a new function. It was previously called AddMesh but has now been renamed for consistency with our naming scheme.

Finally, we call the BuildTerrainBlocks function, which is just a renamed version of the old BuildMeshes method. It is this function that generates the geometry for each terrain block.

```
    // Build the terrain block data itself
    return BuildTerrainBlocks( );
}
```

### CTerrain::Render

The Render method in the CTerrain class has an additional line now. You will remember that the render function essentially just binds the base and detail textures to stages 0 and 1 on the device and then iterates through each terrain block setting its vertex and index buffers on the device and then rendering them. The main difference now is that we must set the terrain's world matrix prior to rendering it since the vertices are now defined in terrain (model) space. The following line has been inserted very near the top of the CTerrain::Render function before we loop through and render each block.

```
// Set this terrain's transformation
m_pD3DDevice->SetTransform( D3DTS_WORLD, &m_mtxWorld );
```

**Note:** The CTerrain class exposes methods that allow the application to set and retrieve the world matrix.

Before each terrain block is rendered we also perform a frustum test on it, just as we did before. When we call the CCamera::BoundsInFrustum function to perform this test, we also pass the world matrix along with the bounding box min and max vectors. Since the terrain block's bounding box is in terrain space it must be converted into world space for the test. The BoundsInFrustum method of CCamera has an optional third matrix parameter that, when used, will cause the function to transform the bounding box extents using the passed world matrix prior to performing the frustum test.

The line of code that is executed before we process each block in the CTerrain::Render function is shown below. It is the first line that is executed in the loop that processes each block. If the terrain block is not visible, the current iteration of the loop is skipped.

```
if ( pCamera && (!pCamera->BoundsInFrustum( m_pTerrainBlocks[i]->m_BoundsMin,
                                            m_pTerrainBlocks[i]->m_BoundsMax,
                                            &m_mtxWorld )) ) continue;
```

## Other Changes to CTerrain

Before leaving this section, we will point out other small changes that have taken place to CTerrain. Most are so small, that to cover the function code again would be redundant.

You will recall from Module I that our CTerrain class registered two static callback functions with the CPlayer object that were used for collision detection purposes. These functions were called UpdatePlayer and UpdateCamera. They were called by CPlayer whenever it wanted to update the position of the player or its attached camera. If the position of either was found to be below the terrain, it would be adjusted so that it lay on the terrain surface (see Chapter Three).

The function involved finding the quad over which the player was positioned and interpolating the height values to find the exact height of the terrain at the new position. If this was higher than the player's position, then the position of the player (or its camera) was updated.

These functions must be updated slightly because we can no longer compare the position of the player against the interpolated vertex heights as they no longer exist in the same space; the player/camera position is in world space and the vertex data of the terrain is in terrain space. In order for these functions to continue working correctly we must add an additional line to the top of each function that transforms the passed player/camera position into terrain space. This is done by multiplying the player position with the inverse world matrix of the terrain. The function will then continue on as usual.

## Updating the CScene IWF Loading Code

In this lab project, the IWF loading code in CScene will be upgraded to support two additional entity types: Terrain and Fog. Since these are both entities, this will amount to nothing more than adding two addition case statements to the CScene::ProcessEntities function.

We will show the ProcessEntities function in a moment. Note that we will snip out the code that processes references we are familiar with (light, external reference, and skybox) although we will leave the case statements in place so you can see the general flow of the function.

## CScene and DirectX Fog

Although we have been able to add fog to our scenes since Chapter Seven of Module I, until this lab project, we were forced to hardcode the fog parameters and generate the results we wanted through tedious trial and error. GILES™ allows us to configure fog and view it in an easy to tweak manner, so it is a good tool for experimenting with our fog parameters (without having to recompile code).

GILES™ saves the fog settings for the scene in an entity chunk with the entity ID seen below. This is accompanied by the GILES™ author ID. We will define this value in CScene.h so that we can easily test for this entity type when searching through the entity vector.

```
#define CUSTOM_ENTITY_FOG       0x204
```

The data area for a fog entity is laid out as shown in the following structure. This structure is also defined in CScene.h and will be used to contain the fog data we fetch from the IWF file.

**Excerpt from CScene.h**

```
typedef struct _FogEntity
{
    ULONG       Flags;                  // Fog type flags
    ULONG       FogColor;               // Color of fog.
    ULONG       FogTableMode;           // Mode to use with table fog
    ULONG       FogVertexMode;          // Mode to use with vertex fog
    float       FogStart;               // Fog range start
    float       FogEnd;                 // Fog range end
    float       FogDensity;             // Density float

} FogEntity;
```

We know from our coverage of DirectX fog in Chapter Seven what most of these values mean and how they contribute to the fog factor, but there are a few members here that need explanation since their meaning is specific to this type of entity.

**ULONG Flags**
These flags can be zero or one of the following modifier flags that help inform us about what type of fog is preferred by the IWF file creator. There are two flags defined in CScene.h that map to these values.

```
#define FOG_USERANGEFOG         0x1
```

If this flag is set it means that if our scene is going to use vertex fog instead of table fog then range based fog is preferred (as is normally the case). This flag is only valid when vertex fog is used as there is no range based pixel fog in DirectX 9.

```
#define FOG_PREFERTABLEFOG      0x2   // Modifier used for FogEntity::Flags member
```

When the flags member of the entity has this flag set, it means the IWF creator would prefer to use table (pixel) fog. This is generally the preferred mode for most applications as it gives the best results (see Chapter Seven of Module I).

**ULONG FogTableMode**
**ULONG FogVertexMode**
These two members of the entity data area define the fog models the artist would like us to use for vertex and table fog modes. They can be set to any of the values you see below. You should understand what these definitions mean and the way the fog factor is computed in each of the fog models since we covered them in Chapter Seven. These values are also defined in CScene.h

```
#define FOGMODE_NONE            0x0         // Mode flags for FogEntity
#define FOGMODE_EXP             0x1         // ''
#define FOGMODE_EXP2            0x2         // ''
#define FOGMODE_LINEAR          0x3         // ''
```

Remember, both the FogTableMode and the FogVertexMode will be set to one of these values. Although an application will typically use only one of these fog modes, specifying the options for each allows us more control over specifying what should be used when one is not supported.

The CScene class itself actually has a 'FogEntity' structure as a member. It is this variable that will be populated with the fog entity data extracted from the IWF file. Below we see the two new members that have been added to CScene to contain any fog data that has been loaded.

```
bool                m_bFogEnabled;    // Is fog enabled or not in this level?
FogEntity           m_FogEntity;      // Storage for our fog setup.
```

When the m_FogEntity structure is populated with fog data from the IWF file, the m_bFogEnabled boolean is also set to true; otherwise it stays set as false. Forgetting about the actual loading code for just a moment, the CScene::Render function has the following lines inserted near the start of the function to set up the fog parameters prior to rendering any objects:

**Excerpt from CScene::Render**

```
if ( m_bFogEnabled )
    {
        // Setup fog parameters
        m_pD3DDevice->SetRenderState( D3DRS_FOGCOLOR  , m_FogEntity.FogColor);
        m_pD3DDevice->SetRenderState( D3DRS_FOGSTART  , *(ULONG*)&m_FogEntity.FogStart  );
        m_pD3DDevice->SetRenderState( D3DRS_FOGEND    , *(ULONG*)&m_FogEntity.FogEnd    );
        m_pD3DDevice->SetRenderState( D3DRS_FOGDENSITY, *(ULONG*)&m_FogEntity.FogDensity);
        m_pD3DDevice->SetRenderState( D3DRS_RANGEFOGENABLE,
                                     (m_FogEntity.Flags & FOG_USERANGEFOG)? TRUE : FALSE );
        m_pD3DDevice->SetRenderState( D3DRS_FOGENABLE, TRUE );
```

The fog color, start and end distances, and density are taken straight from the CScene::m_pFogEntity structure and passed to the pipeline. We also enable the device's fog render state. Notice that we only enable range based fog if the FOG_USERANGEFOG flag has been set in the entity structure.

The final section of the fog render code is shown below. If table fog is preferred then we enable it and disable the vertex fog mode (and vice versa).

**Excerpt from CScene::Render Continued**

```
        // Set up conditional table / vertex modes
        if ( m_FogEntity.Flags & FOG_PREFERTABLEFOG )
        {
            m_pD3DDevice->SetRenderState( D3DRS_FOGTABLEMODE, m_FogEntity.FogTableMode );
            m_pD3DDevice->SetRenderState( D3DRS_FOGVERTEXMODE, D3DFOG_NONE );

        } // End if table fog
        else
        {
            m_pD3DDevice->SetRenderState( D3DRS_FOGVERTEXMODE, m_FogEntity.FogVertexMode );
            m_pD3DDevice->SetRenderState( D3DRS_FOGTABLEMODE, D3DFOG_NONE );

        } // End if vertex fog

    } // End if fog enabled
```

Looking at the above two sections of code it may at first seem as if we are not validating the requested fog techniques on the device to see if they are supported. It looks like we are essentially just extracting the information in the FogEntity structure that was loaded from the file and trying to use them. However, you will see in a moment when we look at the code that loads the fog entity that, after the data

has been loaded into the FogEntity structure, the device is tested and the members are modified to reflect what can and cannot be used on the current device.

## CScene::ProcessEntities (Updated)

This function has been in use since Module I (to parse light entities). Along the way we added support for external reference entities, sky boxes, and most recently, trees. Because these sections of the function have already been covered, the code that parses these entities is not shown again here to compact the listing. In such places where code has been removed we have replaced the code with the line '……snip……'.

As we know, this function is called from CScene::LoadSceneFromIWF after the data from the IWF file has been loaded into the CFileIWF object's internal STL vectors. This function's job is to extract the entity information from that object into a format that can be used by the application. The function sets up a loop to visit each entity in the CFileIWF object's entity vector. Each entity in this vector is of type iwfEntity, which is essentially a structure that contains a world matrix, the entity name, and a data area for storing arbitrary data. It also stores an entity type ID member and an author ID.

```
bool CScene::ProcessEntities( const CFileIWF& File )
{
    ULONG      i, j;
    D3DLIGHT9  Light;
    USHORT     StringLength;
    bool       SkyBoxBuilt = false;
    bool       FogBuilt    = false;

    // Loop through and build our lights & references
    for ( i = 0; i < File.m_vpEntityList.size(); i++ )
    {
        // Retrieve pointer to file entity
        iwfEntity * pFileEntity = File.m_vpEntityList[i];

        // Skip if there is no data
        if ( pFileEntity->DataSize == 0 ) continue;
```

The first thing we do inside the entity loop is get a pointer to the current entity we wish to process. We also test the DataSize member and if it is zero, then the data area is empty and we skip it.

The first entity we test for is the light entity, which is part of the IWF SDK specification. The standard light entity has an entity ID of 0x0010, so if we find an entity with this ID we know we have found a standard light. In the following code you can see that we only continue to parse the light entity if we have not already set up the maximum number of lights we wish to use. As this is not new code to us, the code that extracts the light information from the entity is not shown here.

```
        if ( (m_nLightCount < m_nLightLimit && m_nLightCount < MAX_LIGHTS)
              && pFileEntity->EntityTypeMatches( ENTITY_LIGHT ) )
        {
            //..... snip .....
        } // End if light
```

The light entity is the only entity type we currently process which is part of the core IWF standard. The rest of the entities we will load will be custom entities with the 'GILES' author ID. We can use the iwfEntity object's EntityAuthorMatches method and pass it a char array containing the ID we are looking for. In the following code, AuthorID has been defined at the start of 'CScene.cpp' as a 5 element char array containing the letters 'G','I','L','E','S'. This is the author ID GILES™ assigns to its custom entity chunks. If this function returns true, then the current entity is a GILES™ entity and we will initiate some logic to parse this entity.

```
        else

        if ( pFileEntity->EntityAuthorMatches( 5, AuthorID ) )
        {
            CTerrain * pNewTerrain = NULL;

            SkyBoxEntity SkyBox;
            ZeroMemory( &SkyBox, sizeof(SkyBoxEntity) );

            ReferenceEntity Reference;
            ZeroMemory( &Reference, sizeof(ReferenceEntity) );

            TerrainEntity Terrain;
            ZeroMemory( &Terrain, sizeof(TerrainEntity) );

            FogEntity Fog;
            ZeroMemory( &Fog, sizeof(FogEntity) );
```

Notice in the above code that we instantiate a temporary TerrainEntity structure and a FogEntity structure in addition to the others we previously supported. These structures are used to hold the data we extract.

Next we retrieve a pointer to the entity data area before entering a switch statement that will determine which of the GILES™ custom entities we are dealing with here. Since we have covered the code that processes the reference entity and the sky box entity in previous lessons, the code has been snipped to increase readability.

```
            // Retrieve data area
            UCHAR * pEntityData = pFileEntity->DataArea;

            switch ( pFileEntity->EntityTypeID )
            {
                case CUSTOM_ENTITY_REFERENCE:

                    // ..... snip .....

                    break;

                case CUSTOM_ENTITY_SKYBOX:

                    // ..... snip .....

                    break;
```

Now we add some new content. In the next section of code we add a case statement to test if the current entity has an ID of CUSTOM_ENTITY_TERRAIN. This value is defined in CTerrain.h as:

```
#define CUSTOM_ENTITY_TERRAIN   0x201
```

The value 0x201 is the ID that GILES™ assigns its terrain entity so we know that if this case is executed, it is a GILES™ terrain entity we are processing. Here is the first section of the code that processes the terrain entity.

```
                case CUSTOM_ENTITY_TERRAIN:

                    // Copy over the terrain data
                    memcpy( &Terrain.TerrainType, pEntityData, sizeof(ULONG) );
                    pEntityData += sizeof(ULONG);
```

Earlier in the function we fetched a pointer to the beginning of the entity data area. This data area is laid out in an identical manner to our TerrainEntity structure, so all we have to do is extract the values from the data area into one of these structures. In the above code we copy the first four bytes of the data area which contains the TerrainType value into the TerrainType member of our TerrainEntity structure. We then advance the data pointer past the value we have just extracted so that it is pointing at the beginning of the second piece of data to be extracted.

Before continuing to extract the rest of the data we should test the terrain type value we just extracted. Our CTerrain class encapsulates standard terrains (type 0) so we are currently only interested in parsing those.

In the following code we show a similar technique for copying the data to that which was described above. Each time, we copy the data into the TerrainEntity structure and advance the pointer by the correct number of bytes to point at the next piece of data. The following lines of code show us copying over the terrain Flags, its Width and its Height into their respective members in the TerrainEntity structure.

```
                    // Terrain, per-type processing
                    if ( Terrain.TerrainType == 0 )
                    {
                        memcpy( &Terrain.Flags, pEntityData, sizeof(ULONG) );
                        pEntityData += sizeof(ULONG);

                        memcpy( &Terrain.TerrainWidth,
                                pEntityData, sizeof(ULONG) );

                        pEntityData += sizeof(ULONG);

                        memcpy( &Terrain.TerrainHeight,
                                pEntityData, sizeof(ULONG) );

                        pEntityData += sizeof(ULONG);
```

After the width and height values in the file there is an unsigned short value informing us of the length of the filename string that follows (the heightmap image file). First we copy this value into a temporary

value called StringLength so we know how long the filename that follows will be, then we advance the data pointer to point at the beginning of this string, and finally we copy the filename from the data area into the TerrainDef character array of the TerrainEntity structure. This is shown below.

```
memcpy( &StringLength, pEntityData, sizeof(USHORT) );
pEntityData += sizeof(USHORT);

if ( StringLength > 0 )
   memcpy(Terrain.TerrainDef, pEntityData, StringLength );

pEntityData += StringLength;
```

All strings stored in the entity are defined in this way (i.e., prior to the string data will be an integer value describing the length of the string that follows).

After the filename of the heightmap in the data area, we will find two more strings (with their associated lengths) containing the filenames of the base and detail textures. Once again, the same technique is used for copying into the TerrainTex and TerrainDet members of our TerrainEntity structure.

```
memcpy( &StringLength, pEntityData, sizeof(USHORT) );
pEntityData += sizeof(USHORT);

if ( StringLength > 0 )
memcpy( Terrain.TerrainTex, pEntityData, StringLength );
pEntityData += StringLength;

memcpy( &StringLength, pEntityData, sizeof(USHORT) );
pEntityData += sizeof(USHORT);

if ( StringLength > 0 )
memcpy( Terrain.TerrainDet, pEntityData, StringLength );
pEntityData += StringLength;
```

The final two pieces of data to extract from the data area are the terrain scale (3D vector) and the detail map scale (2D vector). The following code copies these into the TerrainScale and TerrainDetScale members of the TerrainEntity structure.

```
memcpy( &Terrain.TerrainScale, pEntityData,
         sizeof(D3DXVECTOR3) );
pEntityData += sizeof(D3DXVECTOR3);

memcpy( &Terrain.TerrainDetScale, pEntityData,
        sizeof(D3DXVECTOR2) );
pEntityData += sizeof(D3DXVECTOR2);
```

At this point we have extracted all the data from the terrain entity data area and now have it stored in a TerrainEntity structure. We will now generate the terrain object.

First we call the CScene::AddTerrain method to make space at the end of the scene's CTerrain pointer array for an additional pointer. We then allocate a new CTerrain object.

```
                        // Add a new slot ready for the terrain
                        if ( AddTerrain( 1 ) < 0 ) break;

                        // Allocate a new terrain object
                        pNewTerrain = new CTerrain;
                        if ( !pNewTerrain ) break;
```

We send the terrain object the 3D device and the CTextureFormatsEnum object we are using so it can store the pointers internally for its own use. We also call the SetWorldMatrix method and send it the entity's matrix. We then set the data path so that the terrain knows where to find its texture images and heightmap files. Finally, we store the pointer of our new CTerrain object at the end of the scene object's terrain array.

```
                        // Setup the terrain
                        pNewTerrain->SetD3DDevice( m_pD3DDevice, m_bHardwareTnL );
                        pNewTerrain->SetTextureFormat( m_TextureFormats );
                        pNewTerrain->SetRenderMode(GetGameApp()->GetSinglePass());
                        pNewTerrain->SetWorldMatrix((D3DXMATRIX&)
                                                    pFileEntity->ObjectMatrix );

                        pNewTerrain->SetDataPath( m_strDataPath );

                        // Store it
                        m_pTerrain[ m_nTerrainCount - 1 ] = pNewTerrain;
```

We now call the CTerrain::LoadTerrain method passing in the TerrainEntity structure that we have filled out. We know that this is the function that will generate the terrain geometry. We also register the CTerrain player and camera callbacks with the player object (used for collision purposes) and our job is done.

```
                        // Load the terrain
                        if ( !pNewTerrain->LoadTerrain( &Terrain ) ) return false;

                        // Add the callbacks for collision with the terrain
                        GetGameApp()->GetPlayer()->AddPlayerCallback(
                                     CTerrain::UpdatePlayer, pNewTerrain );

                        GetGameApp()->GetPlayer()->AddCameraCallback(
                                           CTerrain::UpdateCamera, pNewTerrain );

                    } // End if 'standard' terrain

                    break;
```

The next and final section of this function is executed if the entity being processed is a GILES™ fog entity. The GILES™ fog entity has an entity ID of 0x204 and we set up a define in CScene.h as shown below:

```
#define CUSTOM_ENTITY_FOG       0x204
```

The first thing we do inside the fog case block is test a boolean variable called FogBuilt. If it was already set to true we skip the entity because it is possible that multiple fog entities might exist in the IWF file and we are only interested in finding and using the first one. Once we have processed this fog entity, the boolean will be set to true and any other fog entities will be ignored.

```
                case CUSTOM_ENTITY_FOG:

                    // We only want one fog setup per file please! :)
                    if ( FogBuilt == true ) break;
                    FogBuilt = true;
```

The data area of the fog entity is laid out in exactly the same way as the FogEntity structure we discussed earlier, so we will extract the data straight into this structure. Here is the remaining code to the function.

```
                    // Retrieve the fog entity details.
                    memcpy( &Fog.Flags, pEntityData, sizeof(ULONG) );
                    pEntityData += sizeof(ULONG);

                    memcpy( &Fog.FogColor, pEntityData, sizeof(ULONG) );
                    pEntityData += sizeof(ULONG);

                    memcpy( &Fog.FogTableMode, pEntityData, sizeof(ULONG) );
                    pEntityData += sizeof(ULONG);

                    memcpy( &Fog.FogVertexMode, pEntityData, sizeof(ULONG) );
                    pEntityData += sizeof(ULONG);

                    memcpy( &Fog.FogStart, pEntityData, sizeof(float) );
                    pEntityData += sizeof(float);

                    memcpy( &Fog.FogEnd, pEntityData, sizeof(float) );
                    pEntityData += sizeof(float);

                    memcpy( &Fog.FogDensity, pEntityData, sizeof(float) );
                    pEntityData += sizeof(float);

                    // Process the fog entity (it requires validation etc)
                    if ( !ProcessFog( Fog ) ) return false;

                    break;

                } // End Entity Type Switch

            } // End if custom entities

        } // Next Entity

        // Success!
        return true;
}
```

Notice that after we have extracted the fog information into the FogEntity structure, we call a new member function of CScene called ProcessFog.

## CScene::ProcessFog

This function is passed the FogEntity structure that contains all the fog settings loaded from the fog entity. This function has the task of testing the requested fog settings to see if they are supported by the device and modifying any settings that are not.

```
bool CScene::ProcessFog( const FogEntity & Fog )
{
    D3DCAPS9  Caps;
    CGameApp *pApp          = GetGameApp();
    bool      bTableSupport = false, bVertexSupport = false;
    bool      bRangeSupport = false, bWFogSupport   = false;

    // Disable fog here just in case
    m_bFogEnabled = false;

    // Validate requirements
    if ( !m_pD3DDevice || !pApp ) return false;

    // Store fog entity
    m_FogEntity = Fog;

    // Retrieve the card capabilities (on failure, just silently return)
    if ( FAILED( m_pD3DDevice->GetDeviceCaps( &Caps ) ) ) return true;
```

The first section of the function shown above starts by setting CScene::m_bFogEnabled to false. We do this just in case fog is not supported. If this is not set to true at some point in the course of this function the render function for CScene will not set any fog parameters. We also store the passed FogEntity structure in the CScene member variable m_FogEntity since this is the structure the CScene::Render function will use to set up the fog parameters for the DirectX pipeline. Then we call the IDirect3DDevice9::GetDeviceCaps method to get the capabilities of the device.

In the next section of code we query four device capabilities and set four local booleans to true or false based on whether they are supported are not. We need to test support for four fog settings:

1. Is table/pixel fog supported on this device?
2. Is vertex fog supported on this device?
3. If vertex fog is supported does it support the more desirable range based vertex fog (instead of view space Z depth fog)?
4. If table fog is supported does it support the use of the W component in its fog calculations (view space Z) or does it use 0.0-1.0 device coordinates (Z buffer coordinates). W based table fog is more desirable for two reasons. First, the Z buffer coordinates have a very non-linear distribution of depth values causing flaky fog effects for objects in the far distance. Second, it is much nicer when using the linear fog model to specify the fog start and fog end distances using view space Z values instead of using 0.0-1.0 normalized device coordinates.

The next section sets the booleans based on the device capabilities. It returns immediately if the device does not support vertex or table fog. If this is the case, our application will not be able to use fog.

```
    // Retrieve supported modes
    bTableSupport  = (Caps.RasterCaps & D3DPRASTERCAPS_FOGTABLE) != 0;
    bVertexSupport = (Caps.RasterCaps & D3DPRASTERCAPS_FOGVERTEX) != 0;
    bRangeSupport  = (Caps.RasterCaps & D3DPRASTERCAPS_FOGRANGE) != 0;
    bWFogSupport   = (Caps.RasterCaps & D3DPRASTERCAPS_WFOG) != 0;

    // No fog supported at all (silently return)?
    if ( !bTableSupport && !bVertexSupport ) return true;
```

Next we perform a few simple comparisons. If the passed TerrainEntity structure had the FOG_PREFERTABLEFOG flag set but the device does not support table fog, we will un-set this flag.

```
    // Strip table flag if not supported
    if ( !bTableSupport ) m_FogEntity.Flags &= ~FOG_PREFERTABLEFOG;
```

If vertex fog is not supported, then table fog must be supported or we would have returned from the function already. Therefore, we make sure that the FOG_PREFERTABLEFOG flag is set. This flag will only be forcefully set if vertex fog is not supported. If vertex fog is supported and the level creator did not specify the preference for table fog, vertex fog will be used.

```
    // Ensure use of table if vertex not supported
    if ( !bVertexSupport ) m_FogEntity.Flags |= FOG_PREFERTABLEFOG;
```

At this point, we have modified the TerrainEntity structure's Flags member such that we know whether we are going to use table fog or vertex fog. In table fog mode we have something else to worry about. If WFog is not supported, we will need to adjust the Fog Start and Fog End members of the terrain entity structure so that they are specified in normalized device coordinates instead of as view space Z values.

The next section of code is executed if table fog is going to be used.

```
    // Fog mode specific setup
    if ( m_FogEntity.Flags & FOG_PREFERTABLEFOG )
    {
        // Ranged is never supported in table mode, strip it
        m_FogEntity.Flags &= ~FOG_USERANGEFOG;

        // If WFog is not supported, we have to adjust these values
         // to ensure that they
        // are in the 0 - 1.0f range.
        if ( !bWFogSupport )
        {
            float fNear = pApp->GetNearPlaneDistance();
            float fFar  = pApp->GetFarPlaneDistance();

            // Shift into range between near / far plane
            m_FogEntity.FogStart -= fNear;
            m_FogEntity.FogEnd   -= fNear;
            m_FogEntity.FogStart /= fFar;
```

```
            m_FogEntity.FogEnd   /= fFar;

        } // End if no WFog support

    } // End if using table mode
```

If the FOG_USERANGEFOG flag is set in the terrain entity structure we remove it since range fogging is only applicable to vertex fog mode. We then test to see if W fog is supported. If it is, then we have nothing else to do since the Fog Start and Fog End members in the terrain entity structure will already be defined in view space (exactly how we should pass them to the pipeline). If it is not supported, we need to map these values into the [0.0, 1.0] range between the near and far planes. We do this by retrieving the near and far plane values currently being used by the CGameApp class via two of its member functions. We then subtract the near plane from the start and end values and divide them by the far plane value. This results in fog start and end values in the [0.0, 1.0] range that describe depth values between those two planes.

Finally if vertex fog is being used instead of table fog, we test to see if the range fog is supported and if not, we make sure the range fog flag in the terrain entity structure is unset.

```
    else
    {
        // If ranged is not supported, strip it
        if ( !bRangeSupport ) m_FogEntity.Flags &= ~FOG_USERANGEFOG;

    } // End if using vertex mode

    // Enable fog
    m_bFogEnabled = true;

    // Success!
    return true;
}
```

At the end of the function we set the scene's m_bFogEnabled boolean to true so the render function knows to use fog.

# Conclusion

A good deal of this workbook has been about consolidating much of what we have learned over the series and upgrading our application framework to support some more advanced constructs. We now have an easy way to describe complex animation sequences for our actor and we have upgraded our code to support multiple terrains, fog, animated actors, trees, and meshes all within the same scene. This is a good place to end the first half of this course because for the remainder of this course, our focus is going to shift almost entirely into the realm of engine design topics. While the goal will not be to design our final game engine just yet (that is our job in Module III), it will be to lay a foundation to do exactly that. We have a lot of extremely important generic game programming topics we need to get through in the coming weeks, so make sure that you feel comfortable with everything we have covered to date.