Lab Project 16.3: BSP Leaf Tree Loading & Rendering

In lab project 16.3, we begin to put together the rendering application that will make use of the information that has been compiled and exported by the new pre-processing tool developed in the previous lab project. In this application we will move back to our integrated spatial partitioning framework which we have been developing in previous lessons and add a new type of tree responsible for handling the solid BSP leaf tree information. Due to the fact that the compilation of the spatial hierarchy is now delegated to an external application, this new tree class will not perform any physical compile process. Instead it will be responsible only for the loading of the tree data from the compiled IWF file and reconstructing that information into a format compatible with our current spatial hierarchy rendering and utility classes.

The Loading Process

Because we have already developed a full scale scene import process within the 'CScene' class, it would be ideal if we could find a way to have this class remain responsible for loading and processing all level geometry and scene entities. If we can achieve this then no large scale modifications will have to be made within the application itself or any part of our existing spatial partitioning framework. In this lab project we will develop a process in which the scene object first processes the selected IWF file in the same manner as it has always done. Any polygon data imported from that file will simply be added to this new tree *loading* object via the 'ISpatialTree' interface as was the case with the tree *compiler* classes. With this polygon data already in place, the new loading class which will be developed in this lesson will simply be responsible for processing the IWF file for a second time. During this stage, the loading code will retrieve and reconstruct **only** the custom BSP tree information contained therein. As far as the application will be concerned, it would appear as if this new tree class has performed a full compilation of the scene geometry in a similar fashion to the quad-tree class for instance. However, all it has really done is to load the hierarchy data from an external file.

Enhanced Visibility State Recording

In addition to developing this new tree loading concept, we will also enhance the mechanism by which we record the visible state of each of the leaves contained in the tree. In previous applications, during the 'ProcessVisibility' update traversal, we visited every leaf in the tree to update its visibility state with a call to its 'SetVisible' method even if that leaf was not visible. In this tree class, we will be adopting a frame counter system similar to that used by our collision detection and response classes, such that the leaf visibility information is invalidated automatically simply by incrementing a centralized counter variable. Whenever a leaf is found to be visible, a new leaf variable will be updated with the current counter value that can simply be tested during rendering to see if it was marked as visible at any point in this frame of the rendering loop. If the 'SetVisible' call was not made during this frame, then the leaf's internal copy of the counter will be out of date and as such we can take this as a sign that the leaf is not visible. This new mechanism unfortunately prevents the application from making a call to the default 'IsVisible' method of any leaf due to us no longer always updating the 'm_bVisible' member of the 'CBaseLeaf' class. As a result we will need to derive a new leaf class from this base leaf, so that we can

override this behaviour and once again restore this function to a working state. This situation will be discussed in more detail as we move on to discuss the new 'CBSPTreeLeaf' class.

As a direct result of using this new counter mechanism however, we no longer have to visit every single leaf in the spatial hierarchy just to mark it as invisible. As you might image this should have a dramatic effect on the performance of the BSP tree rendering process specifically because it often has many more leaves than the previous types of spatial hierarchy we have developed.

This should be a very interesting use of our existing spatial hierarchy technology. So, without further ado, let us move on to cover the implementation of this new type of tree loader object and its associated support classes and routines.

The CBSPTreeNode Class

As we know, the BSP leaf tree is a *binary* partitioning construct that is designed to separate space into two parts – one portion on either side of a separating plane. This principal is of course identical to that of the kD-tree concept we are already familiar with.

Recall that the node structures within both spatial partitioning schemes are required to maintain similar information. Such information includes a separating plane, front and back child nodes and an optional child leaf. There are also some additional pieces of data that are common to both systems – such as the node's bounding box extents – that serve as the basis for the various rendering and traversal optimizations we have integrated in the past.

If we take a look at the declaration for the new 'CBSPTreeNode' class we should notice that it bears a striking resemblance to the 'CKDTreeNode' class we have previously developed.

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

    // Public Variables for This Class
    D3DXPLANE       Plane;              // Splitting plane for this node
    CBSPTreeNode *  Front;              // Node in front of the plane
    CBSPTreeNode *  Back;               // Node behind the plane
    CBSPTreeLeaf *  Leaf;               // Leaf may be stored here
    D3DXVECTOR3     BoundsMin;          // Minimum bounding box extents
    D3DXVECTOR3     BoundsMax;          // Maximum bounding box extents
    signed char     LastFrustumPlane;   // The frame-to-frame coherance index.

    // Public Members Omitted
};
```

Due to the fact that this node structure is almost identical to that used by both the kD-tree and BSP node tree, we will only briefly recap on each of the member variables declared here.

D3DXPLANE Plane
This member describes the separating plane used in the creation of this node. In the case of the polygon-aligned BSP tree, this will be oriented such that it matches the plane of at least one polygon in the scene. In this new rendering application, the plane data is pulled from our combined plane array – stored in the import file – and placed into the D3DXPLANE typed member that we have relied upon throughout. We will examine this procedure shortly.

CBSPTreeNode * Front
Each node in the BSP tree partitions the scene into pieces which represent the space contained in both its front and back halfspaces. If there are further polygons available to be selected in front of the current node, then our compiler will have recursed into the front list and continued to generate new nodes from that data. The first of these nodes would be attached to this member in exactly the same manner with which we are familiar. In contrast to our compiler tool, our rendering application does not combine the concept of attached child leaves and nodes into one member variable and as such only a child node can be attached here. This is similar to the previous run-time tree types we have implemented prior to this, represented with a type that contains a pointer to another 'CBSPTreeNode' object.

CBSPTreeNode * Back
This member is similar to the previous in that it stores a pointer to any child node that might exist in the back halfspace of the current. Again this is identical in principal to the kD and BSP node tree types we have implemented in the past.

CBSPTreeLeaf * Leaf
Should this be a terminal node, this member will store the child leaf that should be attached here. In the earlier lab projects dealing with spatial hierarchies, we found that *every* terminal node would have a leaf attached to it. In this application we are implementing the run-time portion of the **solid** leaf BSP tree. Recall in lab project 16.2, there were cases in which a particular area of space within the scene was found to be solid. In these cases, no leaf was created or inserted into the tree. These leaves were simply replaced with an indicator that allowed us to identify this fact. As a result, when reconstructing the BSP tree in this application, there may be cases where this member is assigned a value of NULL that is used to signify the same thing. More details will be provided on this subject as we move into covering the actual reconstruction of the hierarchy from file.

D3DXVECTOR3 BoundsMin
This member stores the value that describes the minimum extents of the node's bounding box. Recall that this bounding box will be large enough to contain every node, leaf and polygon that is found to exist anywhere beneath the current node.

D3DXVECTOR3 BoundsMax
Like the previous, this member stores part of the data required to describe the node's axis-aligned bounding box. In this case however, this member stores the *maximum* extents of the box that describes each of the elements contained in this node's subtree.

signed char LastFrustumPlane
We have encountered and discussed this member in previous lessons. Recall that it is used during the 'ProcessVisibility' traversal process when testing the node bounding box against the camera's frustum

planes. This member stores the last frustum plane found to intersect this node's parent bounding box. This is an optional value that can be passed into the camera's 'BoundsInFrustum' function to allow it to optimize the order in which it tests the frustum planes during the processing of a hierarchy such as this.

With each of the member variables discussed, it should be clear that this node class does not declare anything that we have not encountered before with the kD-tree node type. Similar to the 'CKDTreeNode' class, this exposes only a single method. Let us take a look at this now.

CBSPTreeNode::SetVisible

The only method defined by this node class is the 'SetVisible' function. As we know, this function is traditionally called during the 'ProcessVisibility' traversal in each of our previous tree types. This implementation of the BSP tree is no different in that regard. It accepts a single Boolean parameter that specifies whether or not this node and its children are visible, this function then passes that information on to any applicable children. At the top of the function we can see that if there is a leaf stored in this node, that leaf's own 'SetVisible' function will be called – passing in the same visibility status – before returning. If there was no leaf stored here, then this recursive procedure calls the 'SetVisible' function of both its front and back child nodes should either be available. This has the effect of traversing through the tree and updating the visibility status of every leaf contained anywhere beneath the initial node on which this function has been called.

```
void CBSPTreeNode::SetVisible( bool bVisible )
{
    // Set leaf property
    if ( Leaf  ) { Leaf->SetVisible( bVisible ); return; }

    // Recurse down front / back if applicable
    if ( Front ) Front->SetVisible( bVisible );
    if ( Back  ) Back->SetVisible( bVisible );
}
```

As we can see, this simple recursive function remains unchanged from each of the four node classes we have created within the previous spatial hierarchy types.

The CBSPTreeLeaf Class

This class provides the data structures required to represent the leaf items contained within the spatial hierarchy. Again the concepts involved in attaching objects of this type to the tree should be very familiar to us already. However, there is a key difference between the implementation of the leaf concept used in previous spatial hierarchies and that of this new BSP loading class. In each of the previous tree types, recall that we used instances of the 'CBaseLeaf' class exclusively within the tree structure. This base class provides much of the functionality required to store the various pieces of information – such as the polygons and objects contained within that leaf – in addition to the functionality required for rendering much of that data.

As we move forward with the BSP leaf tree concepts, the leaf structure will be required to store and process much more information that is specific to this type of tree. As a result, within the implementation of this spatial hierarchy type, we will begin to create a new leaf class that *derives* from

‘CBaseLeaf’. This is done in order to extend that base with the additional members and functionality required in both this lesson and the next.

With this in mind, let us take a look at this new leaf class declaration before moving on to the initial methods that will be exposed.

```
class CBSPTreeLeaf : public CBaseLeaf
{
public:
    // Constructors & Destructors for This Class.
             CBSPTreeLeaf( CBSPTreeLoader * pTree );
    virtual ~CBSPTreeLeaf( );

    // Public Virtual Functions for This Class (from base).
    virtual bool    IsVisible  ( ) const;
    virtual void    SetVisible ( bool bVisible );

    // Public Variables for This Class
    ULONG           m_nVisCounter;
};
```

As you can see, there is only one member variable declared by this class outside of those inherited from the base ‘CBaseLeaf’ class that we are already familiar with. The purpose of this variable is described below.

ULONG m_nVisCounter
This member relates to the new visibility counting mechanism mentioned earlier in this lesson. Should the leaf be visible in any given render call, this variable will be updated such that it contains the same value as that currently stored within the main tree object. If at any point the leaf is found to be outside of the viewing frustum, this member will *not* be updated and will automatically become out of date as the main counter has been incremented. Because the tree’s counter is incremented every frame, if the value of this member differs from that of the tree object at any point, we know that this leaf was found not to be visible within **this** frame of the render loop.

With this new visibility counting scheme we must alter the way that the ‘CBaseLeaf’ class detects and reports this leaf’s visibility status to the user. This is achieved by overloading the virtual ‘IsVisible’ method such that when the application requests the leaf’s visibility information through a pointer to the ‘ILeaf’ superclass, the function defined in this new ‘CBSPTreeLeaf’ will be called. In addition, we must also provide an overload of the ‘SetVisible’ function. This method is *not* declared within the ‘ILeaf’ interface and should never be called by the application. However, we must overload the ‘CBaseLeaf::SetVisible’ function in order to perform the additional processing that is now required.

CBSPTreeLeaf::CBSPTreeLeaf( )

The first function we need to discuss is the single constructor for this class. This constructor accepts a single parameter as input. This parameter is a pointer to the tree object into which this leaf will be attached. Recall from our earlier discussion of the ‘CBaseLeaf’ class, this *parent* information is placed into an internal member variable and is used to allow the leaf object to access information stored within

the tree itself. Both the member variable and the code to store the input parameter are inherited from this class's base. As a result, we must ensure that the base class constructor is also called, passing to it the same information that this constructor received. The easiest way to achieve this is to simply call the relevant 'CBaseLeaf' constructor by using the initialization list portion of the 'CBSPTreeLeaf' constructor shown below.

```
CBSPTreeLeaf::CBSPTreeLeaf( CBSPTreeLoader * pTree ) : CBaseLeaf( pTree )
{
    // Reset / Clear all required values
    m_nVisCounter   = 0;
}
```

With the information inherited from 'CBaseLeaf' fully initialized by the base class constructor, we can then simply continue to initialize those members added by this class. In this lab project, the only variable we need to reset here is the 'm_nVisCount' member. This is set to '0' in order to ensure that every leaf starts its life in an *in*visible state.

CBSPTreeLeaf::SetVisible

Earlier we talked a little bit about the new mechanism by which we are recording information about whether or not a leaf is visible. This scheme uses a counter variable that describes whether or not the leaf was marked as visible within the most recent call to the tree class's 'ProcessVisiblity' method. In this function and the next, we can see just how this is achieved. Recall that this is the first spatial hierarchy type in which we are deriving a new class in order to customize some of the core functionality provided by the base 'CBaseLeaf'. Because we want to add an additional mechanism by which the visibility status of the leaf is recorded we need to overload the base class' 'SetVisible' function.

In the code block that follows we have included additional code that updates the member variable responsible for recording the most recent frame in which this leaf was found to be visible. If this leaf was specified as being visible then the current visibility frame counter value is requested of the parent tree object to which this leaf is attached. The value returned is subsequently stored in the leaf's 'm_nVisCounter' member variable. If a value of 'false' was passed to this method's single parameter, then we simply reset the visibility counter to 0. Although in this application we will never explicitly call the 'SetVisible' method when a leaf is **not** visible – due to our new updated counter mechanism automatically invalidating such leaves – we include the latter case in order to ensure that the system exhibits the correct behaviour should we need to do so in the future.

```
void CBSPTreeLeaf::SetVisible( bool bVisible )
{
    // Update our current vis counter
    if ( bVisible )
    {
        // Store the current visibility counter
        m_nVisCounter = ((CBSPTreeLoader*)m_pTree)->GetVisCounter();

    } // End if visible
    else
    {
        // Reset the visibility counter
```

```
        m_nVisCounter = 0;

    } // End if not visible
```

With the ‘m_nVisCounter’ member variable either having been updated with the current frame counter, or reset to a value of 0 depending on the state of the ‘bVisible’ parameter, we must now pass on this message to the base class implementation of this function. This is done to allow the ‘CBaseLeaf’ version of this method to perform any necessary steps for adding this leaf to the tree’s internal visible leaf array as well as populating the appropriate buffers prior to rendering.

```
    // Call base class implementation
    CBaseLeaf::SetVisible( bVisible );
}
```

Given this relatively simple logic it should be clear that the ‘m_nVisCounter’ member of any given leaf will only be updated to match that of the tree if this method has been invoked in any specific visibility update traversal. Because the *tree’s* internal visibility counter is automatically incremented with each subsequent call to the ‘ProcessVisibility’ function in every frame, this of course means that if the leaf was not found to be visible at any point, the ‘m_nVisCounter’ variable maintained by that leaf will no longer match. This effectively allows us to mark every leaf as invisible at the start of each frame simply by incrementing that single counter value maintained by the main tree object. This negates the need to explicitly call the ‘SetVisible’ method for each leaf, passing a value of ‘false’ during the visibility traversal as we have done in the past. We will take a closer look at the changes we can make to the ‘ProcessVisibility’ method of the main tree class to improve its efficiency a little later in this lesson.

Due to the fact that we still want the application to be able to query the visibility state of each leaf, we must also override the ‘IsVisible’ method to take this new frame counter mechanism into account.

CBSPTreeLeaf::IsVisible

Should the leaf’s counter variable have been updated in a call to its ‘SetVisible’ method then logically the value stored in the leaf’s visibility counter should match that maintained by the associated tree object. Of course, this will only be the case for the duration of time until the next call to ‘ProcessVisibility’ is made. If the leaf is not subsequently marked as visible in that frame as well, then the value stored in the leaf’s ‘m_nVisCounter’ variable will be out of date. This removes the need for us to traverse every part of the tree simply to set a leaf as *not* being visible.

Given these facts, the ‘IsVisible’ method has been overloaded in this class to return a status of ‘true’ **only** if the leaf was set to visible in this frame of the applications main rendering loop – or more accurately the most recent call to the tree’s ‘ProcessVisibility’ method. This is done by comparing the visibility counter value stored here, to the value currently maintained by the parent tree object we set in the class constructor. If these two values match, then we know that the leaf has been updated in the current frame and the comparison operation shown below will result in a value of true being returned. If this is not the case, then the function will return false.

```
bool CBSPTreeLeaf::IsVisible( ) const
{
```

```
    return (m_nVisCounter == ((CBSPTreeLoader*)m_pTree)->GetVisCounter());
}
```

With our newly overridden leaf tree and node classes implemented in their entirety, let us now begin to examine the core derived tree class that we will by using in this lab project.

The CBSPTreeLoader Class

Like each of the spatial tree classes we have developed in the past, this new class is derived from ‘CBaseTree’ from which we inherit a large portion of the required storage, management and rendering functionality. The only task that each of these spatial hierarchy classes has essentially been responsible for is to populate the leaf, node and polygon arrays maintained by the base class itself. As we know this was previously achieved by taking the polygon data that has been loaded and passed in to the tree class and compiling that information into a given type of spatial hierarchy. In the case of lab project 16.3, the compilation process has already taken place and we are simply reconstructing that tree into a format compatible with our already existing spatial hierarchy utility and rendering classes. As a result, the majority of the code that we will be implementing within this class revolves around the loading and interpreting of the BSP tree information contained within the source IWF file and simply storing it in the appropriate base class container members. With this in mind, let us take a look at the new ‘CBSPTreeLoader’ class declaration to see how we might integrate such a concept into the existing hierarchy system.

```
class CBSPTreeLoader : public CBaseTree
{
public:
    // Constructors & Destructors for This Class.
    virtual ~CBSPTreeLoader( );
             CBSPTreeLoader( LPDIRECT3DDEVICE9 pDevice, bool bHardwareTnL,
                             LPCTSTR FileName );

protected:
    // Typedefs, Structures and Enumerators.
    struct iwfNode
    {
        long        PlaneIndex;
        D3DXVECTOR3 BoundsMin;
        D3DXVECTOR3 BoundsMax;
        long        FrontIndex;
        long        BackIndex;
    };

    // Protected Variables for This Class
    CBSPTreeNode * m_pRootNode;         // The root node of the tree
    LPTSTR         m_strFileName;       // File to load

    CBaseIWF       m_FileLoader;        // The IWF parsing object
    iwfNode       *m_pFileNodes;        // Node data loaded from file
    D3DXPLANE     *m_pFilePlanes;       // Plane data loaded from file.
    ULONG          m_nFileNodeCount;    // Number of nodes loaded from file
    ULONG          m_nFilePlaneCount;   // Number of planes loaded from file
```

```
    // Protected Functions Omitted

};
```

There are several new members declared here that we have not yet encountered in any of the spatial hierarchy classes that we have previously developed. Each of these member variables are outlined below.

CBSPTreeNode * m_pRootNode
The ‘m_pRootNode’ member stores a pointer to the root ‘CBSPTreeNode’ object of the constructed spatial hierarchy. This is the entry point into the tree and is used to start any traversal operation as we know. This member will be assigned during the tree reconstruction step in the ‘BuildTree’ function discussed shortly.

LPTSTR m_strFileName
This member stores a duplicated copy of the path and file name for the file to be imported. This filename string should reference the IWF file that has been compiled and exported by the pre-processing tool developed in lab project 16.2. This member is populated in this class’s constructor and later used by the ‘Build’ function during the file import step (covered shortly).

CBaseIWF m_FileLoader
Due to the fact that this tree type does not perform a tree compilation process and instead simply loads the data from the IWF file, we need access to the file data. For this we use the ‘CBaseIWF’ class provided to us by the ‘libIWF’ import library. The ‘m_FileLoader’ member stores an object of this type. Put simply, this object will be used to perform all of the actual parsing of the IWF file and its structure.

By using a series of callback functions, the ‘CBaseIWF’ object will notify us whenever it encounters a chunk of a type within the file that we are interested in. With the file automatically positioned at the beginning of that applicable data, we can simply read the information using the stream functions provided by the ‘CBaseIWF’ class. For more information on this class and its methods, refer to appendix A in this chapter. We will see how this can be used to help us load the custom tree data as we move on to discuss the overloaded ‘Build’ function in addition to the registered callbacks that we must supply in order to read that information.

iwfNode * m_pFileNodes
During the import of the tree information contained in the IWF file, there are cases where we must load the data into temporary structures. The file data that describe the nodes within the BSP tree is just such a case. As we know, each node item stored within the file has a number of dependencies. These include the planes that are contained in the file’s combined plane array, the leaf objects that may be attached to the node and of course the front and back child nodes. At the point in which we are loading the data for any given node, we may not yet have imported any plane or leaf items that may be attached to that node. More importantly than either of these is the fact that, until the entire node array has been loaded, the build procedure will not have access to the front and back child node information and will be unable to rebuild the physical tree structure. As a result, we must import the node data into a temporary area until such time as every node has been loaded. Once this has been done, we will then have full access to each node that is contained in the tree in order for us to reconstruct it.

Because of the fact that the structure of the node data stored within the file is different to that of the run-time application, and uses an indexing scheme to reference the applicable node, leaf and plane information, we must create a new structure that will hold the file's information temporarily. Let us take a quick look at the iwfNode structure we have defined in order to achieve this.

struct iwfNode

- *long PlaneIndex*

  As with each of the variables declared within this 'iwfNode' structure, the data stored in this member is loaded directly from the IWF file that is being processed. In particular, this member stores an index into the file's combined plane array that describes the plane on which this particular node lies. This information will be used to retrieve the correct plane data during the reconstruction of the tree into a format suitable for this application.

- *D3DXVECTOR3 BoundsMin*

  This member stores the value that describes the minimum extents of the node's bounding box. Recall that this bounding box will be large enough to contain every node, leaf and polygon that is found to exist anywhere beneath it in the tree.

- *D3DXVECTOR3 BoundsMax*

  Like the previous, this member stores part of the data required to describe the node's axis-aligned bounding box. In this case however, this member stores the *maximum* extents of the box that describes each of the elements contained in this node's subtree.

- *long FrontIndex*

  This member represents the index of the item attached to the front side of the current node stored within the appropriate array. Recall from our earlier discussion of lab project 16.2 that this can either be another child node or a leaf. If the sign of the index is positive then this member references a child node. If it is negative, this signifies that a leaf is attached to this side of the node. Our run-time tree hierarchy is constructed in a slightly different manner, providing additional leaf-nodes to which we attach the physical leaf data, simplifying the traversal process. This is yet another reason why ensuring that the node data is fully loaded and stored in a temporary array before reconstructing the tree is useful to us.

- *long BackIndex*

  The 'BackIndex' member is identical in principal to the aforementioned 'FrontIndex'. The only difference here is that this member is used to reference any leaf or node child that may be attached to the *back* of the current node. Also recall that in the solid tree – such as that compiled by the new pre-processing tool – this member may also store a value equal to that defined by the 'BSP_SOLID_LEAF' constant if the space behind this node is to be considered solid. It is

important to bear this in mind because we will need to recognize this situation when reconstructing the tree hierarchy.

With each of the individual elements in the temporary node structure defined, let us now continue on to discuss the remainder of the member variables declared within the 'CBSPTreeLoader' class.

D3DXPLANE * m_pFilePlanes
This member stores the array of plane data that was compiled and stored in the scene IWF file. Due to the fact that the plane information is stored in a separate linear block from the node data, this member also acts as a temporary container much like the 'm_pFileNodes' array. This array will be populated during import and will be used during the reconstruction of the spatial hierarchy as the basis for the plane data stored at each node. Recall that the node structure within the file stores an index to a specific plane within the combined plane array stored in the file. Since this array will represent that same plane data it should be used as the source for the plane referenced by the 'PlaneIndex' value stored in each temporary 'iwfNode' structure.

ULONG m_nFileNodeCount
This member stores the total number of nodes loaded from file and subsequently stored in the array referenced by the 'm_pFileNodes' member variable. This information is not used directly during the reconstruction of the tree, but it can be used for the purposes of validation to ensure that node data was actually loaded from the source file. This information might also be used to check each node's front and back index variables such that we return gracefully if any index exceeds the boundaries of the array, indicating that the node data stored in the file is malformed or corrupted.

ULONG m_nFilePlaneCount
As with the previous variable, this member stores the total number of planes loaded from file and subsequently stored in the 'm_pFilePlanes' array. Again, this information is not used directly during the reconstruction of the tree but it might be used to provide an extra layer of validation to protect against file corruptions or invalid information.

As we can see, with the exception of the 'm_pRootNode' variable, each of the members contained within this *loader* class exist primarily for the purpose of handling the import and processing of the scene hierarchy data. This information will then be used to reconstruct the hierarchy in a manner compatible with the 'ISpatialTree' interface concept we have been developing. Now that we have a rough understanding of each of the new class member variables, let us now move on to discuss the various methods of this class that are responsible for performing these operations.

CBSPTreeLoader::CBSPTreeLoader( )

We are already familiar with the steps involved in setting up a constructor for a new tree type that has been derived from 'CBaseTree'. As in each of our earlier tree classes, the constructor must accept both a valid Direct3D device, in addition to a boolean flag indicating whether hardware transform and lighting can and should be used when rendering any scene geometry. These two pieces of information must be passed to the base class constructor to allow any vertex and index buffer resources to be created using the correct parameters. In this new 'CBSPTreeLoader' class, this is achieved in a similar manner to those we have previously implemented, by passing these two parameters to the base constructor using

the initialization list portion of the function. The only additional piece of information that is required specifically by our new BSP tree loading class is the name of the file from which the tree data should be imported. This is passed by the application as the third argument to this constructor.

The body of this constructor is very simple and is responsible for initializing the class member variables with their default values. In this particular constructor we must also make a copy of the filename specified by the application. We do this using the '_tcsdup' runtime function in the same way as we have done many times before. It is important that we make a copy of this string here rather than simply storing the specified string pointer in order to ensure that the memory referenced by the 'FileName' parameter is not altered or released before we get a chance to begin importing and building the tree information.

```
CBSPTreeLoader::CBSPTreeLoader( LPDIRECT3DDEVICE9 pDevice, bool bHardwareTnL,
                                LPCTSTR FileName ) :
                                CBaseTree( pDevice, bHardwareTnL )
{
    // Clear required variables
    m_pRootNode       = NULL;
    m_pFileNodes      = NULL;
    m_pFilePlanes     = NULL;
    m_nFileNodeCount  = 0;
    m_nFilePlaneCount = 0;
    m_nVisCounter     = 0;

    // Make a copy of the file name
    m_strFileName = _tcsdup( FileName );
}
```

Once an object of this type has been instantiated, and the relevant pieces of information initialized and stored, the application can then instruct that object to begin building the spatial hierarchy information.

CBSPTreeLoader::GetVisCounter

The 'GetVisCounter' function is a publically accessible class method that allows other methods and objects to gain access to the tree's current visibility 'call' counter. This method is used by the 'SetVisible' and 'IsVisible' methods of the new 'CBSPTreeLeaf' class in order to retrieve the current counter value that has been incremented by a call to 'ProcessVisibility'. This function simply returns the value currently stored in the 'm_nVisCounter' member variable and takes no further action.

```
ULONG CBSPTreeLoader::GetVisCounter( ) const
{
    return m_nVisCounter;
}
```

CBSPTreeLoader::Build

The 'Build' function in the 'CBSPTreeLoader' class is a virtual function declared initially by the base 'ISpatialTree' interface. In each of the tree classes we have developed to date, the application has used this function to trigger the actual tree compilation process after the scene data has been added. However,

as we mentioned earlier, this class will not actually implement any kind of compile behavior. At this point, the information associated with the tree has already been compiled by our pre-processing tool and written to file. As a result, the 'Build' function of this class is primarily responsible only for setting up and calling the various import and reconstruction procedures that will create and populate the appropriate tree data structures.

One thing that you may notice as we move through the code in this class is that it only imports a small portion of the source IWF file specified in the class constructor. At no point does this class ever load or manipulate any type of scene polygon data for instance. This is due to the fact that the application will already have imported much of this information on our behalf. With the polygon data in particular, these will have already been loaded from the file in the 'CScene' class, and added to the member arrays inherited by this class using the 'CBaseTree::AddPolygon' function. With the scene responsible for loading much of the physical level data, this class needs only to import and process the BSP tree information that was exported by the tool developed in lab project 16.2. Any other information that may be stored in the file will simply be ignored. Thanks to the 'CBaseIWF' class exposed by the libIWF library, this is actually made very simple.

As defined by the 'ISpatialTree' super-class, the 'Build' method accepts a single parameter as input. This parameter is named 'bAllowSplits' and is a simple Boolean value that traditionally instructed the compiler as to whether the resulting polygon data should be split against the node planes or left whole. In the case of the 'CBSPTreeLoader' class, the tree and its associated polygon data have already been constructed and written to file based on the parameters specified in the pre-processing step. As a result this parameter will be ignored and the geometry will instead simply be used in its existing form.

```
bool CBSPTreeLoader::Build( bool bAllowSplits /* = true */ )
{
    try
    {
        // Open the file
        m_FileLoader.Open( m_strFileName, CBaseIWF::MODE_READ );

        // Set up the author ID for custom chunk reading
        m_FileLoader.SetAuthorID( "BSPC1", 5 );
```

Earlier we mentioned the 'm_FileLoader' member which is an instance of the 'CBaseIWF' class exported by the libIWF import library. This class handles the majority of the processing and navigating of an IWF file's structure and provides us with an easy means to gain access to the data stored within that file. In order to have this object process a specific file, the first thing we must do in this function is to open the file that we are interested in importing. This is achieved by calling the 'Open' method of the 'm_FileLoader' object passing in the name of the file to be processed. Recall that when an instance of this 'CBSPTreeLoader' class is created, the name of the IWF file that we are loading is passed to the constructor and duplicated into the 'm_strFileName' member variable. Therefore, this is the string that is passed as the first argument to the 'Open' method of the 'm_FileLoader' object. In addition, we must also instruct this object that the file should be opened for the purpose of *reading* data rather than writing. To this end, we also pass the 'CBaseIWF::MODE_READ' enumerator item as the second argument to this same function.

During the call to the ‘Open’ method, the file will be tested to ensure that it is of a format supported by the import library. Should this be the case, any appropriate file header information will be processed automatically and the ‘m_FileLoader’ object will be initialized such that we can begin reading data immediately. If an error occurred – such as the file being of an unrecognized format – the ‘CBaseIWF’ class will throw an exception that must be handled by our application such that we can return a failure code and potentially exit from the application if necessary. For this reason, the entire file loading logic is wrapped in a ‘try’ block that is used to this end.

In addition to specifying the source file that we would like this object to process, we must also inform the file object of the information that identifies any custom chunk data contained in the file as belonging to this application. Recall from our earlier discussions of the author ID with respect to entities and other custom data stored with the IWF, each of these items have a configurable signature that can be specified by the exporting application. This is done to ensure that only the application(s) that are specifically aware of its format ever try to import it. When loading custom chunks from the IWF file using the ‘CBaseIWF’ class, we can request that the object notifies us of these custom data chunks by specifying this signature with a call to the ‘SetAuthorID’ method of the ‘CBaseIWF’ class. This method accepts two parameters that require an arbitrary byte or character array – which denotes the author signature to verify against – and its associated length in bytes. Assuming that we pass in the same combination of signature bytes to this function as was specified during export, the IWF processing object will step into these custom data areas and correctly process the information instead of ignoring it.

In order to inform the ‘CBaseIWF’ object of the types of custom chunk we are interested in, we make use of the provided ‘RegisterChunkProc’ method. This function accepts the chunk ID value as the first of its three parameters. Recall that when exporting the plane, node and leaf data in the pre-processing tool we used the following three defines to identify each type of custom chunk written to file:

```
#define CHUNK_CTM_PLANES    0x2000
#define CHUNK_CTM_NODES     0x2001
#define CHUNK_CTM_LEAVES    0x2002
```

By re-using these same identifier values we are instructing the IWF processing object that we are interested in being notified whenever these three chunk types are encountered within the file. The means by which the IWF library notifies us of these occurances is by using the same callback mechanism we have used many times before in our lab project framework. By passing a function pointer to the second parameter of the ‘RegisterChunkProc’ method for each type of chunk, those static callback functions will be executed after the file has been positioned at the start of the appropriate data area.

As we know, due to the fact that these types of callback functions are static, it is also useful for the calling function to specify a context pointer. This is often a piece of custom data that can be used to pass any required information on to the callback procedure. In this application we pass in the pointer to the ‘CBSPTreeLoader’ class instance which is currently being processed. This context value is passed to the third parameter of this function to which we simply specify the ‘this’ pointer in each case. In doing so we allow each callback function that is executed to access the members of this specific tree object such that it might store any information that may have been loaded.

```
        // Load leaf and node data
        m_FileLoader.RegisterChunkProc( CHUNK_CTM_PLANES, ReadBSPPlanes, this );
```

```
        m_FileLoader.RegisterChunkProc( CHUNK_CTM_NODES , ReadBSPNodes, this );
        m_FileLoader.RegisterChunkProc( CHUNK_CTM_LEAVES, ReadBSPLeaves, this );
```

The callback functions that we register with the file loader object must be static and should each have a function signature that matches exactly with the one expected by this class. The function pointer typedef for the chunk callback procedures is shown below.

```
typedef void (*CHUNKPROC)(LPVOID pContext,CHUNKHEADER& Header,LPVOID pCustomData);
```

As we can see, each callback procedure will be passed three parameters and is not expected to provide any sort of return value. Given this specification we might imagine any given chunk procedure to be declared similar to the following:

```
static void MyProc( LPVOID pContext, CHUNKHEADER& Header, LPVOID pCustomData );
```

We will shortly discuss the specific details of each of the chunk callback functions that we registered here and will discuss the function parameters and their purpose in each case.

Now that we have registered the callback functions for each of the custom chunk types we would like to be informed of we can instruct the ‘m_FileLoader’ object to begin the processing of the currently open file. This is achieved by calling the ‘ProcessIWF’ method which will proceed to step through the various chunks in the file and call the appropriate procedure should any registered chunk type be encountered. Once the file has been processed in its entirety by this method and the registered callback procedures, we can then close the open file handle with a simple call to the ‘m_FileLoader.Close()’ method.

```
        // Load the additional BSP specific data
        m_FileLoader.ProcessIWF( );

        // Close the file
        m_FileLoader.Close();

    } // End Try Block

    catch ( ... )
    {
        // Complete Failure
        return false;

    } // End Catch Block
```

At this point, the registered chunk procedures should have loaded any node, leaf and plane data required for us to reconstruct the BSP tree. If there was no relevant tree data found to exist in the file, or a problem occurred for whatever reason, we should not continue with the reconstruction process. To this end we must verify that the tree data was indeed loaded by checking both the value contained in the ‘m_nFileNodeCount’ variable, in addition to the size of the ‘m_Leaves’ STL vector inherited from the base ‘CBaseTree’ class. If either of these members indicates that no relevant data was loaded then we return a failure code back to the calling function. Again, we will see the importance of checking these two variables specifically when we move on to discuss the actual chunk procedures registered earlier.

Should the required data have been imported successfully, we are now able to put together the various components to construct the spatial hierarchy. This is achieved with a call to the 'BuildTree' function which is a method of this class. We have encountered this function several times in previous lessons whereby it was traditionally responsible for the actual compilation and processing of the scene data in order to compile a particular type of spatial hierarchy. With the information having already been loaded from file, this function will simply be responsible for taking that information and rebuilding the tree structure in a compatible format. We will examine this function in a little while, but for now all that is important at this stage are the arguments that are being passed in to this function.

```
    // No BSP tree data loaded?
    if ( m_nFileNodeCount == 0 || m_Leaves.size() == 0 ) return false;

    // Reconstruct the tree structure from that loaded
    BuildTree( &m_pFileNodes[0], NULL );
```

The first parameter declared by the 'BuildTree' function is the current *source* node of the type 'iwfNode' (our temporary file based structure). Due to the fact that the node data is currently stored in a flat linear array, each of which contain only indices to inform us of the various parent/child relationships, it might not be instantly apparent exactly where to begin with the reconstruction process. Recall from our earlier coverage of the centralized node array in lab project 16.2 however, we always allocated the root node such that it would be located in the first element of that array. Therefore, using the temporary node information we loaded from the source IWF file, we begin this process by passing in that first element to the 'BuildTree' function. The second parameter is the current *destination* node that we will be constructing from the loaded source data during the recursive process. Since we have no root node at this point we simply pass NULL to this first call of the 'BuildTree' method. We will see how this information is used later in this lesson.

During the import of the spatial hierarchy data, recall that we used temporary arrays in order to store the node and plane information ready for processing. Now that we have fully reconstructed the BSP tree, this information is no longer required. As a result, we call a new method of this class called 'ReleaseFileData'. This function simply releases any memory associated with those temporary arrays to ensure that we don't consume any more memory than is necessary during the lifetime of the application.

```
    // Release the file data we had loaded.
    ReleaseFileData();

    // Allow our base class to finish any remaining processing
    return CBaseTree::PostBuild( );
}
```

At this stage we have imported, built and populated the arrays and hierarchy structures for each of the node and leaf items required. This is all of the information we need in order to begin using the BSP tree we compiled separately. With all of the pre-requisite information now available, we finally call the 'PostBuild' function of the base tree class to allow it to perform any additional processing on the tree data we have built. Recall that the 'PostBuild' procedure is responsible for constructing the vertex and index buffers that will be used during the rendering of the scene. The result of the 'PostBuild' function is returned directly to the calling function in order to signify the final success or failure of the build operation.

With the primary build interface function now covered, let us move on to take a look at the chunk procedure callbacks we registered in order to allow us to load the relevant components of the BSP tree hierarchy.

CBSPTreeLoader::ReadBSPPlanes

The first registered chunk procedure that we come to is the 'ReadBSPPlanes' function. As with all such registered procedures this function is defined as being static and must conform precisely to the function signature we discussed in the coverage of the 'Build' function earlier. This callback function accepts three parameters. The first of these is a void pointer that is used to inform each callback function of the context in which it is being executed. We have seen this type of callback system used many times throughout previous lessons, so the concept of a callback context should be nothing new to us. When registering this callback function with the 'CBaseIWF' object in the 'Build' function, recall that we passed a value of 'this' to the context parameter of the 'RegisterChunkProc' method. This of course means that the underlying object to which this first parameter will point is an instance of the 'CBSPTreeLoader' class that we are currently in the process of initializing. In this function, the loader class instance referenced by the context parameter will be the destination for any plane data that may be encountered in this part of the file. It is generally a good idea to cast this pointer to the expected type early on in the callback function to ensure that the code remains as simple and clean as possible.

The second and third parameters are not used by this application and are reserved mostly for use by the internal functionality of the 'CFileIWF' class exported by the libIWF library. As a result we will not spend too much time discussing their purpose. Put simply however, these parameters are used to pass additional file chunk information to the various callback procedures registered with the system. The 'Header' parameter for instance references a structure that contains various chunk properties such as the type of chunk being processed and the length in bytes of the chunk data area. For more information on the various pieces of additional chunk information available here, please refer to the IWF specification and SDK which should be available to you in your class supplemental download area.

At the point at which any registered chunk procedure is called by the IWF processing class, the file being processed should be positioned at the start of the relevant chunk data area within the file. What this essentially means is that we can begin reading the data relevant to this particular type of chunk immediately on entering this function. In order to do so however, we must gain access to the 'CBaseIWF' object instance that is being used to load the tree information. At the start of this function therefore, we cast the specified context pointer to that of our 'CBSPTreeLoader' class instance that we are populating, and then retrieve from that object a pointer to the 'm_FileLoader' member that specifies the 'CBaseIWF' object for the file we are currently reading from.

```
void CBSPTreeLoader::ReadBSPPlanes( LPVOID pContext, CHUNKHEADER& Header,
                                    LPVOID pCustomData )
{
    ULONG i;

    // Retrieve context pieces.
    CBSPTreeLoader * pLoader = (CBSPTreeLoader*)pContext;
    CBaseIWF *pFile = &pLoader->m_FileLoader;
```

Now that we have access to both the source file object and the destination 'CBSPTreeLoader' class instance, we can begin to load the plane data from the file. We achieve this directly through the file / stream handling methods exposed by the 'CBaseIWF' class which include functions such as 'Read', 'Write' and 'Seek'. A full list of each of these methods and their usage can be found in appendix A of this chapter's workbook. The first item we come to is a simple four byte 'unsigned long' that specifies how many planes – in total – are stored within the file. We read this value with a call to the 'CBaseIWF::Read' method, passing in a pointer to the 'm_nFilePlaneCount' member of the 'pLoader' object into which the plane count will be loaded directly, bypassing the need for a temporary variable.

Next on the agenda is to allocate enough room in our loader class' temporary 'm_pFilePlanes' array to store every plane contained within this chunk. The number of planes is of course described by the 'm_nFilePlaneCount' variable we initialized prior to this step and is used to correctly size the array with the new operator. Before we move on, this new array is cleared with a call to the Win32 'ZeroMemory' function such that we begin with a sensible set of default values.

```
    // Retrieve the Plane Count
    pFile->Read( &pLoader->m_nFilePlaneCount, sizeof(ULONG) );

    // Allocate storage for the planes
    pLoader->m_pFilePlanes = new D3DXPLANE[ pLoader->m_nFilePlaneCount ];
    ZeroMemory( pLoader->m_pFilePlanes,
                pLoader->m_nFilePlaneCount * sizeof(D3DXPLANE) );
```

With the temporary plane array allocated, the actual plane information can now be loaded and stored ready for the reconstruction step which occurs once the file has been completely processed. The code used to achieve this is shown below in which a loop is created that iterates through each of the planes in the new temporary plane array. These planes are used as the destination into which each subsequent plane contained within this file chunk is loaded and stored. Similar to loading the simple unsigned long value we encountered earlier, this is achieved with a call to the 'CBaseIWF::Read' method passing in a pointer to the destination plane structure stored in the temporary array. In addition we also pass a value which indicates the number of bytes that must be read from the file in order to load the plane in its entirety. This process is repeated for each plane indicated by the 'm_nFilePlaneCount' member until all planes have been loaded from the file and stored in the temporary 'm_pFilePlanes' member array.

```
    // Read Planes
    for ( i = 0; i < pLoader->m_nFilePlaneCount; i++ )
    {
        D3DXPLANE * pPlane = &pLoader->m_pFilePlanes[i];

        // Load the plane information from file
        pFile->Read( pPlane, sizeof(D3DXPLANE) );

    } // Next Plane
}
```

At this stage, the 'm_pFilePlanes' array contained within the 'CBSPTreeLoader' class has been allocated and fully populated with that data exported by the pre-processing tool developed in lab-project 16.2. The plane data is left in its combined form to ensure that the plane indices maintained by each

node remain valid. Of course, in order for the plane information to be of any use to us, we must also import the node data. Therefore, let us now examine the chunk callback procedure responsible for loading that information.

CBSPTreeLoader::ReadBSPNodes

As we observed with the 'ReadBSPPlanes' function, the static callbacks we must implement in order to load the custom tree data from the IWF file are relatively straight forward. The same is true when loading the node data into our temporary array of 'iwfNode' structures.

The majority of the code in this function is identical to that of the plane loading callback we discussed previously so we will not spend much time discussing the details here. This function simply reads the value describing the total number of nodes into the 'm_nFileNodeCount' member variable and allocates the temporary node array in much the same way as we did in the 'ReadBSPPlanes' callback. Due to the fact that we have defined the interim 'iwfNode' structure using the same format and layout of the node items contained in the file, we then simply read the node data one item at a time storing that information directly into each element in that temporary node array.

```
void CBSPTreeLoader::ReadBSPNodes( LPVOID pContext, CHUNKHEADER& Header,
                                   LPVOID pCustomData )
{
    ULONG i;

    // Retrieve context pieces.
    CBSPTreeLoader * pLoader = (CBSPTreeLoader*)pContext;
    CBaseIWF *pFile = &pLoader->m_FileLoader;

    // Retrieve the Node Count
    pFile->Read( &pLoader->m_nFileNodeCount, sizeof(ULONG) );

    // Allocate storage for the nodes
    pLoader->m_pFileNodes = new iwfNode[ pLoader->m_nFileNodeCount ];
    ZeroMemory( pLoader->m_pFileNodes,
                pLoader->m_nFileNodeCount * sizeof(iwfNode) );

    // Read Nodes
    for ( i = 0; i < pLoader->m_nFileNodeCount; i++ )
        pFile->Read( &pLoader->m_pFileNodes[i], sizeof(iwfNode) );

}
```

CBSPTreeLoader::ReadBSPLeaves

The 'ReadBSPLeaves' callback is by far the most complex of the tree import procedures implemented here. This is primarily due to the fact that there is much more information to be loaded, processed and stored in each leaf than in either of the other two cases. Similar to the other two callbacks defined within this class, this method is responsible for loading the leaf data from file. However, unlike those procedures, this method will load and populate an instance of a 'CBSPTreeLeaf' object rather than a temporary data structure. As we know, the reason we loaded the node and plane data into temporary arrays was due to the dependency that exists between them and the fact that we could not be certain

which of the two data chunks would be encountered first within the IWF file. In the case of the leaf however, all pre-requisite information has already been loaded by the scene import process, or is contained directly within the file leaf chunk we are reading from in this function.

As with each of the previous two functions, this is a static callback that accepts a void context pointer along with any appropriate chunk and custom data information neither of which is used by this application. Again, we cast the specified context pointer to that of our 'CBSPTreeLoader' class instance that we are populating, and retrieve the 'm_FileLoader' member that specifies the 'CBaseIWF' object for the file we are currently processing.

```
void CBSPTreeLoader::ReadBSPLeaves( LPVOID pContext, CHUNKHEADER& Header,
                                    LPVOID pCustomData )
{
    ULONG       i, j, LeafCount, PolygonCount, PortalCount, ReservedCount;
    D3DXVECTOR3 vecMin, vecMax;
    long        PolygonIndex;

    // Retrieve context pieces.
    CBSPTreeLoader * pLoader = (CBSPTreeLoader*)pContext;
    CBaseIWF *pFile = &pLoader->m_FileLoader;
```

Now that we have access to the source and destination objects we can begin to read the leaf data from file. The first item we come to is a simple four byte 'unsigned long' that specifies how many leaves – in total – are stored within the file. We read this value with a call to the 'CBaseIWF::Read' method, passing in a pointer to the local 'LeafCount' variable declared at the top of this function. We don't need to store this information directly as the tree class's internal leaf count will be incrememented as each leaf is finally added. If no leaves are found to be contained within the file we simply return.

Now we have the information that describes the total number of leaves in the tree, we can reserve the 'm_Leaves' member STL vector of the tree object using the vector's 'reserve' method. This will prevent any subsequent 'CBaseTree::AddLeaf' call from having to resize / reallocate the leaf array.

```
    // Retrieve the Leaf Count
    pFile->Read( &LeafCount, sizeof(ULONG) );
    if ( LeafCount == 0 ) return;

    // Allocate storage for the leaves
    pLoader->m_Leaves.reserve( LeafCount );
```

With this step completed, we can now begin to read the leaf information from file. The 'CHUNK_CTM_LEAVES' chunk stores all of the leaves in the tree together in one linear block. The size of this block of leaves is indicated by the 'LeafCount' variable that was read from the file a moment ago. At this point therefore we create a loop that will execute for this same number of iterations as each leaf is loaded.

The first thing we must do within this loop is to allocate a new 'CBSPTreeLeaf' object that will eventually be inserted at some point into the tree hierarchy. This leaf object will be used to store the information read directly from the file rather than into a temporary structure as with the node type. Remember from our earlier discussion that each leaf must have access to a pointer of the tree object in

which it is contained. This is done in the following code – in a similar manner to the previous compilation classes – by passing the tree object's pointer into the single parameter of the 'CBSPTreeLeaf' constructor. Unlike previous applications however, the 'ReadBSPLeaves' function is static and, as a result, we do not have access to the 'this' pointer. If you refer back to the earlier discussion of the 'Build' function, you will remember that when we registered each callback the 'this' pointer was specified as the callback context at that time. At the start of this function we cast the context pointer to the correct tree class type and copied it over into the 'pLoader' variable. Therefore, we can instead simply pass in the 'pLoader' variable to the leaf's constructor.

What follows this allocation is the reading of the various pieces of leaf data contained in the file. The majority of this information is loaded into temporary variables due to the fact that the actual leaf properties will be updated as we perform the various operations such as adding polygon data or setting the leaf's bounding box via the 'AddPolygon' or 'SetBoundingBox' methods inherited from the 'CBaseLeaf' class. The first two values to be loaded here are the minimum and maximum bounding box vectors that describe the absolute world space extents of the polygon data contained in this leaf. These are stored in the local 'vecMin' and 'vecMax' variables. Following these vectors is a value that is currently reserved for a later lab project. This is a single unsigned long that will eventually be used to store an index into the main visibility array that we will be constructing in the next lesson. For now we simply seek over this value.

The next two values contained in the file describe the total number of elements for both the polygon and portal index data that we will shortly encounter in the file. Again, the 'PortalCount' variable is not used by this application and is reserved for the visibility compiler we will build in the next lesson. However, it is important that we read this information in case we attempt to import a file built by the later version of our pre-processing tool such that we can skip over any portal data that may be contained in this file for this leaf. We will discuss the means by which this is achieved a little further on, but for now just know that these two variables describe the total number of indices maintained by the leaf for both the polygons and portals that are attached to this leaf.

The final of these count variables is the 'ReservedCount' item. Again this is not used by any of our current lab projects, but this reserved concept can be used by an application to store custom information along with the leaf data. The value retrieved by this read operation will describe the amount of space reserved for this custom data that will allow us to skip that portion of the leaf should it not be of any interest to us. Again we will talk about this more specifically further on in this function's coverage.

```
    // Read Leaves
    for ( i = 0; i < LeafCount; i++ )
    {
        CBSPTreeLeaf * pNewLeaf = new CBSPTreeLeaf( pLoader );

        // Read initial part of structure
        pFile->Read( &vecMin, sizeof(D3DXVECTOR3) );
        pFile->Read( &vecMax, sizeof(D3DXVECTOR3) );
        pFile->Seek( sizeof(ULONG) ); // Seek over PVSIndex
        pFile->Read( &PolygonCount, sizeof(ULONG) );
        pFile->Read( &PortalCount, sizeof(ULONG) );
        pFile->Read( &ReservedCount, sizeof(ULONG) );
```

With this first part of the leaf now loaded and stored we can begin to initialize the actual leaf object we created in the initial stages of this loop. The first thing we do here is to set the leaf's bounding box using the local 'vecMin' and 'vecMax' variables that were populated in the first two read operations for this leaf. Once this has been done we then begin to link any appropriate polygon data to the leaf object itself. To do this, we create a loop that will execute for the total number of iterations specified by the value stored in the 'PolygonCount' variable we loaded earlier. Recall that this value describes the total number of indices stored in this particular leaf that indicate which polygons we should attach here. At each step we then read in a single 4 byte value from the file that describes the index of the polygon to be attached to this leaf. Due to the fact that the polygon data has been loaded by the scene class and added to the 'CBSPTreeLoader' polygon array in advance we can simply retrieve the pointer stored at that element in the inherited 'm_Polygons' array, and pass that pointer directly into the new leaf's 'AddPolygon' method. This will proceed to take that pointer and update the leaf's internal polygon list to store the specified polygon.

```
        // Store bounding box details in the new leaf
        pNewLeaf->SetBoundingBox( vecMin, vecMax );

        // Read the face indices
        for ( j = 0; j < PolygonCount; ++j )
        {
            // Read the index into the polygon array
            pFile->Read( &PolygonIndex, sizeof(long) );
            if ( PolygonIndex < 0 ) continue;

            // Add the specified polygon to the leaf
            pNewLeaf->AddPolygon( pLoader->m_Polygons[ PolygonIndex ] );

        } // Next Leaf
```

Once this loop has completed the required number of iterations, we should have loaded each of the polygon indices contained in the file for this leaf, and added the physical polygon pointers to the new leaf object that we are building. With this operation completed, we can now move on to process the remaining items stored in the file.

Due to the fact that we are not interested in any exported portal index information in this application, the final thing we must do is to skip over any such information that *may* be contained in the file. Although the pre-processing tool developed in lab project 16.2 does not export these leaf portal indices, it is advisable that we do this regardless in case the source IWF file was not exported by that version of the compiler. Notice below that we also seek over the actual reserved data area. The size of this area is determined by the value read and stored in the local 'ReservedCount' variable which describes the number of DWORD sized chunks (32 bits / 4 bytes) of reserved data that is stored here. Again, no reserved data is written to file in lab project 16.2 and the 'ReservedCount' variable should contain a value of 0. However, as with the portal indices, we seek over any data that may have been indicated.

Now that we have read all of the data for this leaf, we can add it to the tree. However, we do not yet have access to the full spatial hierarchy and are unable to attach this leaf to any node that might reference it. As a result, we simply pass the new leaf into a call to the tree class' 'AddLeaf' method – inherited from the 'CBaseTree' class – that will proceed to store the leaf in the main centralized array. We can later retrieve this pointer when we reconstruct the tree hierarchy in the 'BuildTree' function.

```
        // Skip the reserved data areas
        pFile->Seek( PortalCount * sizeof(long) );
        pFile->Seek( ReservedCount * sizeof(long) );

        // Add this leaf to our leaf array
        pLoader->AddLeaf( pNewLeaf );

    } // Next Leaf
}
```

At this stage, the file should be positioned at the start of the *next* leaf ready to be loaded in the next iteration of this loop. We continue this entire process until all of the indicated leaves have been loaded and stored in the tree's centralized leaf array.

Once this loop has read and stored the leaves contained within the file and, assuming that the node and plane data has already been loaded at this stage, we now have all the information we need to be able to reconstruct the tree. Let us now take a look at how this is achieved.

CBSPTreeLoader::BuildTree

There have been several references to the reconstruction of the BSP tree throughout our coverage of the lab project 16.3 implementation. In previous applications, we developed a private 'BuildTree' function which served as the main recursive procedure in which the tree construction was undertaken. In lab project 16.3, this function has a similar role in that it must recursively build the spatial hierarchy. However, this time it is solely responsible for taking any information that has been previously loaded by the 'Build' function and piece it together such that it now exists in the appropriate format for use by our run-time rendering application.

This function accepts two parameters. The first of these indicated by the 'pFileNode' parameter is the current interim file node that will be used as the source for building a 'CBSPTreeNode' that will be used in the final spatial hierarchy for this tree object. The second 'pNode' parameter is the destination 'CBSPTreeNode' object that is to be populated with any relevant information. Recall from the earlier discussion of the 'Build' function that the first time that this 'BuildTree' method is called, a value of NULL is passed to the 'pNode' parameter. This is used to indicate that we are constructing the root node of our final spatial hierarchy. As a result, the first task undertaken by this function is to allocate a new root node and store the resulting pointer in the tree's 'm_pRootNode' member variable. Of course, this will only happen the first time that this method is invoked due to each subsequent call being passed an already instantiated destination node object.

After creating the root node, or simply using the destination node that was passed, we begin populating this node with the simple bounding box and plane information indicated by the interim 'iwfNode' structure passed to this iteration of the recursive procedure. The first of these is the separating plane on which the node lies. Recall that we imported the combined plane array from the file and stored these in a temporary 'm_pFilePlanes' member array. Due to the fact that the final 'CBSPTreeNode' class stores a physical plane item, the plane indicated by the file node must be extracted from this array and stored directly into the destination node object. The final two values we update in the initial stages of this

function are the ‘BoundsMin’ and ‘BoundsMax’ values which are taken directly from the imported node structure.

```
bool CBSPTreeLoader::BuildTree( iwfNode * pFileNode,
                                CBSPTreeNode * pNode /* = NULL */ )
{
    D3DXVECTOR3 vecMin, vecMax;

    // First time in?
    if ( !pNode )
    {
        // Allocate a root node
        if ( !(pNode = new CBSPTreeNode) ) return false;
        m_pRootNode = pNode;

    } // End if no node specified

    // Build node data from that loaded from file
    pNode->Plane     = m_pFilePlanes[ pFileNode->PlaneIndex ];
    pNode->BoundsMin = pFileNode->BoundsMin;
    pNode->BoundsMax = pFileNode->BoundsMax;
```

With these simple properties updated, this function now proceeds to process and create the child node and leaf items where necessary. The first case we come to is the creation of any applicable front child. Because we are always guaranteed to have a front child whenever we are dealing with a node that does **not** store a leaf, we first allocate a new ‘CBSPTreeNode’ item which will represent the child in front of the current node. The pointer to this new node item is then stored into the ‘Front’ member of the node we are currently processing. Once we have allocated this new front node item, this function then tests to determine whether the interim file node indicates that a child *node* or child *leaf* is attached to the front side of the node at this level. The first case we come to is one in which a child *node* has been indicated by the ‘FrontIndex’ member of the imported node structure, in which we would find a positive index value. If this turns out to be the case, then there is nothing further for us to do with the current node and we simply recurse into the new front child node by making a further call to the ‘BuildTree’ function. At this stage we pass in as the first of two parameters the imported node that was specified by the ‘FrontIndex’ value of the current ‘iwfNode’, as well the node destination node we allocated a moment ago. Should we enter this initial ‘if’ statement, then this process would begin again such that the entire subtree of the current node is built and attached to the front side of the current destination ‘CBSPTreeNode’ object.

```
    // Allocate new node in front
    pNode->Front = new CBSPTreeNode;
    if ( !pNode->Front ) return false;

    // Node or leaf in front?
    if ( pFileNode->FrontIndex >= 0 )
    {
        // Build this new front node
        if ( !BuildTree( &m_pFileNodes[ pFileNode->FrontIndex ], pNode->Front ) )
            return false;

    } // End if node in front
```

The alternate case that we have to deal with is one in which there is a child *leaf* attached to the front side of the current node. This is indicated by the 'FrontIndex' value of the imported node having a **negative** value. Because the initial 'if' statement simply test for any positive value including zero, we can ensure that we are dealing with a leaf simply by using the 'else' keyword. If we drop into this 'else' case we must of course attach a 'CBSPTreeLeaf' item to the front of this node. However, recall that our run-time spatial hierarchy design requires that a leaf-node always be inserted above any given leaf in the tree. Due to the fact that we always explicitly allocated a front node earlier in this function, this is taken care of automatically. By simply attaching the indicated leaf to the **font child** rather than the current node, we can always ensure that this parent leaf-node exists. Notice that we first retrieve the pointer to the correct leaf from the tree's existing leaf array using the leaf indexing logic we discussed in lab project 16.2. Recall that because the value for any indices available for describing a leaf in the file's front or back index members is in the range of -1 and below we must add 1 to the 'FrontIndex' member here before flipping its sign. By doing so this value is converted back into a valid array index in the range of 0 and above.

```
    else
    {
        // Build a leaf
        CBSPTreeLeaf * pLeaf =
                       (CBSPTreeLeaf*)m_Leaves[ abs(pFileNode->FrontIndex + 1) ];
        if ( !pLeaf ) return false;

        // Store pointer to leaf in the node
        pNode->Front->Leaf = pLeaf;

        // Store the leaf's bounding box in the node
        pLeaf->GetBoundingBox( vecMin, vecMax );
        pNode->Front->BoundsMin = vecMin;
        pNode->Front->BoundsMax = vecMax;

    } // End if leaf in front
```

With the front side of this node and the entire front subtree fully reconstructed, this function now turns its attention to the back. We perform the initial steps in an identical fashion to that of the front simply substituting each of the relvant front indices and pointer variables with those specifying the information attached to the back of each node. In the same way as before, we first allocated a new back child 'CBSPTreeNode' object for the current node and then recurse into the back subtree should the value of the 'BackIndex' member of the imported node indicate that a child *node* should be attached here.

```
    // Allocate new node behind
    pNode->Back = new CBSPTreeNode;
    if ( !pNode->Back ) return false;

    // Node or leaf behind?
    if ( pFileNode->BackIndex >= 0 )
    {
        // Build this new back node
        if ( !BuildTree( &m_pFileNodes[ pFileNode->BackIndex ], pNode->Back ) )
            return false;
```

```
    } // End if node in front
```

Once again, the alternate case is one in which an attached child *leaf* is indicated by the imported node data. In this case, we perform exactly the same steps as discussed previously. However, there is an additional consideration that must be taken with this type of compiled BSP tree information when processing the information for the back side of any given node. This is of course when the back halfspace of that node is describing an area of solid space.

If we think back to the development of the BSP leaf tree compilation process, solid leaves were not physically created or inserted into the tree and instead a value equal to that of the 'BSP_SOLID_LEAF' defined constant was stored in the 'BackIndex' member of the node. To this end, the code within this else clause first tests to see whether the current file node's 'BackIndex' member contains this value. If it does then of course we must also signify this 'solid' area within our current spatial hierarchy framework. We achieve this by simply storing a value of 'NULL' in the back child of the current destination node after releasing the child node that we allocated earlier. We could alternately create a back leaf-node that simply stores a value of 'NULL' in its 'Leaf' pointer member. However, the former method that we have chosen to employ in this lab project is slightly more efficient in both processing and memory overhead and has no significant drawbacks.

If the space behind the node is not describing a solid area of space, then we simply attach the leaf indicated by the imported node in the same manner as with any front child leaf.

```
    else
    {
        // Solid leaf?
        if ( pFileNode->BackIndex == BSP_SOLID_LEAF )
        {
            delete pNode->Back;
            pNode->Back = NULL;

        } // End if solid leaf
        else
        {
            // Retrieve the leaf specified
            CBSPTreeLeaf * pLeaf =
                          (CBSPTreeLeaf*)m_Leaves[ abs(pFileNode->BackIndex + 1) ];
            if ( !pLeaf ) return false;

            // Store pointer to leaf in the node
            pNode->Back->Leaf = pLeaf;

            // Store the leaf's bounding box in the node
            pLeaf->GetBoundingBox( vecMin, vecMax );
            pNode->Back->BoundsMin = vecMin;
            pNode->Back->BoundsMax = vecMax;

        } // End if empty back leaf

    } // End if leaf behind

    // Success!
    return true;
```

```
}
```

At this stage, from the point of view of the first level of recursion, both the front and back subtrees will now have been fully reconstructed and attached to the front and back member pointers of the root node. Hopefully it should be clear that no physical compilation has really been undertaken here, we have simply taken the information stored in the file and converted it into the applicable format ready for our application to use.

CBSPTreeLoader::ReleaseFileData

Recall that the 'ReleaseFileData' method is called by the 'Build' function after the tree data has been loaded and reconstructed. Due to the fact that this information is only required during import – and would simply consume memory unnecessarily – each of these file data arrays should be released once the hierarchy has been built. This function is responsible for clearing these temporary file data arrays and resetting any appropriate values for the possibility of later use.

```
void CBSPTreeLoader::ReleaseFileData()
{
    // Destroy arrays
    if ( m_pFileNodes  ) delete []m_pFileNodes;
    if ( m_pFilePlanes ) delete []m_pFilePlanes;

    // Clear Variables
    m_pFileNodes      = NULL;
    m_pFilePlanes     = NULL;
    m_nFileNodeCount  = 0;
    m_nFilePlaneCount = 0;

}
```

CBSPTreeLoader::ProcessVisibility

In this method we begin to see the visibility call counting scheme from the point of view of the tree class. The 'ProcessVisibility' function is almost identical to those we have implemented in each of our spatial hierarchy classes so far. The only addition here is that we increment the 'm_nVisCounter' member of the *tree* class before we begin the recursive 'UpdateTreeVisibility' process. It is important that we do this first, because we need to ensure that each of those leaves subsequently found to be visible are updated with a value that will remain constant until the *next* call to this function. If this value was to be incremented after the update process, then the visibility counter value stored in each leaf will become out of date the moment we stepped out of the 'UpdateTreeVisibility' call.

```
void CBSPTreeLoader::ProcessVisibility( CCamera & Camera )
{
    CBaseTree::ProcessVisibility( Camera );

    // Increment the visibility counter for this loop
    m_nVisCounter++;

    // Start the traversal.
```

```
    UpdateTreeVisibility( m_pRootNode, Camera );
}
```

CBSPTreeLoader::UpdateTreeVisibility

Once again, the 'UpdateTreeVisibility' method defined by this new tree class is almost identical to those we have already encountered. However, there is one minor alteration to this function that has a significant impact from an execution point of view.

As we can see, with the exception of substituting the type of the 'pNode' parameter to that of our new node class, and including this method in the new 'CBSPTreeLoader' namespace, the initial part of this function remains unchanged. Each of the parameters are declared and used in the same manner as before, and the frustum culling result code is retrieved in the same way. Once we have retrieved this frustum value, we then come to the conditional 'switch' statement that is also part of each of the previous implementations.

```
void CBSPTreeLoader::UpdateTreeVisibility( CBSPTreeNode * pNode, CCamera & Camera,
                                           UCHAR FrustumBits /* = 0x0 */ )
{
    CCamera::FRUSTUM_COLLIDE Result =
                             Camera.BoundsInFrustum( pNode->BoundsMin,
                                                     pNode->BoundsMax, NULL,
                                                     &FrustumBits,
                                                     &pNode->LastFrustumPlane );

    // Test result of frustum collide
    switch ( Result )
    {
```

The first case we come to when determining if the current node's bounding box is fully within the frustum is that of the 'FRUSTUM_OUTSIDE' case. The only difference in this function can be found in here. Recall that this result is returned whenever that bounding box is *completely* outside of all of the camera's frustum planes and as a result cannot possibly seen. In addition we know that none of its children can possibly be seen either because of the fact that a node's bounding box will be large enough to contain every child node beneath it in that branch of the tree.

If you recall the earlier implementations of this function, we previously called the node's 'SetVisible' method, passing in a value of 'false' in this *outside* case. This function would then traverse this node's subtree simply in order to visit every node and leaf that existed as a child of the current node, marking each leaf encountered as invisible. With our new visibility call counter mechanism, this process of updating each leaf's visibility status when it is *not* visible is no longer required. As a result, this case now simply returns from the function without taking any additional action.

As we move forward and add concepts such as the 'Potential Visibility Set' – covered in the next lesson – this adjustment could save us a significant amount of additional processing in a highly occluded scene. However, even in the case of simple frustum culling, this technique can be employed in order for us to remove a large portion of the traversal operation we were previously required to perform during the 'ProcessVisibility' pass.

```
        case CCamera::FRUSTUM_OUTSIDE:
            // Node (and all its children) are not visible
            return;
```

The remaining cases in this switch statement remain unaltered from our previous implementations. In the ‘FRUSTUM_INSIDE’ case we must continue to recurse through the subtree of the current node in order to set each child leaf as visible. In the ‘FRUSTUM_INTERSECT’ case we must continue to perform the frustum tests for any child node until we find a node that is either contained within or is completely outside of the frustum. This is of course unless this is a leaf node, in which case the leaf must be set to a visible state to ensure that the visibility status information and rendering buffers are updated correctly for that leaf.

```
        case CCamera::FRUSTUM_INSIDE:
            // Node is totally visible
            pNode->SetVisible( true );
            return;

        case CCamera::FRUSTUM_INTERSECT:
            // We need to resolve this further, unless this is a leaf
            if ( pNode->Leaf )
            {
                pNode->SetVisible( true );
                return;
            } // End if leaf
            break;

    } // End Switch
```

As before, the final block of code is only ever executed if the bounding box of the current node was found to be intersecting the frustum planes. If this was the case then further consideration needs to be taken and as a result we must traverse down the front and back of this node.

```
    // The remaining case (FRUSTUM_INTERSECT) means we need to test further
    if ( pNode->Front ) UpdateTreeVisibility( pNode->Front, Camera, FrustumBits );
    if ( pNode->Back  ) UpdateTreeVisibility( pNode->Back, Camera, FrustumBits );
}
```

Additional CBSPTreeLoader Routines

There are several tree related support routines that we have developed in the past that allow the application to perform operations such as the collection of a list of leaves that are intersected by either an axis aligned bounding box or a ray, and the drawing of debug information. Due to the large similarities between the basic principals of both the BSP tree and the kD-tree, the ‘CBSPTreeLoader’ class duplicates several of these functions taken directly from the ‘CKDTree’ class. Because we have covered the kD-tree support methods in earlier lessons, we will not spend any time covering them again here.

The list below outlines those functions which are direct copies from the ‘CKDTree’ class that have simply been altered to be declared within the ‘CBSPTreeLoader’ namespace and also to make use of the new ‘CBSPTreeNode’ class.

```
bool CollectLeavesAABB    ( LeafList & List, const D3DXVECTOR3 & Min,
                            const D3DXVECTOR3 & Max );
bool CollectAABBRecurse   ( CBSPTreeNode * pNode, LeafList & List,
                            const D3DXVECTOR3 & Min, const D3DXVECTOR3 & Max,
                            bool bAutoCollect = false );
bool CollectLeavesRay     ( LeafList & List, const D3DXVECTOR3 & RayOrigin,
                            const D3DXVECTOR3 & Velocity );
bool CollectRayRecurse    ( CBSPTreeNode * pNode, LeafList & List,
                            const D3DXVECTOR3 & RayOrigin,
                            const D3DXVECTOR3 & Velocity );
void DebugDraw            ( CCamera & Camera );
bool DebugDrawRecurse     ( CBSPTreeNode * pNode, CCamera & Camera,
                            bool bRenderInLeaf );
bool GetSceneBounds       ( D3DXVECTOR3 & Min, D3DXVECTOR3 & Max );
```

Scene Class Modifications

With the new BSP tree loading class fully implemented, the only task that remains is to make any modifications to the application code such that this new spatial hierarchy class is utilized. Because of the way that the ‘ISpatialTree’ concept has been developed, switching to other types of tree within the application is trivial. We have observed on several occasions how we can alternate between an oct-tree, quad-tree and kD-tree simply by altering a single line in the ‘CScene’ class such that the relevant type of object is instantiated before the scene is loaded. The new ‘CBSPTreeLoader’ class is integrated into the application in exactly the same way. With this in mind let us look at the minor modification that we must make in the scene class’ ‘LoadSceneFromIWF’ function in order to make use of the compiled solid leaf BSP tree.

CScene::LoadSceneFromIWF

The following code block shows a small portion of the ‘LoadSceneFromIWF’ method that we should already be extremely familiar with. The only line of code that has changed here is marked in **bold**. We can see that when instantiating the spatial partitioning tree object we have simply created an object of the type ‘CBSPTreeLoader’, storing the returned pointer in the scene’s ‘m_pSpatialTree’ member variable. Recall that this member is of the type ‘ISpatialTree’ which allows the application to access the tree functionality without necessarily having to be aware of the specifics of the selected tree class itself.

When instantiating this type of spatial partitioning tree we pass in the common parameters required by all types of tree that we have developed. These are the Direct3D device that was registered with the

scene during application startup, along with the Boolean flag describing whether hardware transformation and lighting is being used. These two parameters are used by the underlying 'CBaseTree' class to create and populate the vertex and index buffers used in the rendering of the geometry registered with the tree. The third and final parameter is the one that is specific to the 'CBSPTreeLoader' class. Here we simply pass in the filename that was also passed to the 'LoadSceneFromIWF' method by the application. This means that both the scene geometry and the custom tree information will be loaded from the same file. Whilst this is convenient it is not a requirement. Due to the fact that the BSP tree loader class opens and processes the file independently of the scene class, it would be possible to have the scene geometry and BSP tree information stored in separate files. In this case we would simply pass in an alternate filename to the loader class constructor.

```
    ...
    ...

    // File loading may throw an exception
    try
    {
        // Allocate our spatial partitioning tree of the required type
        m_pSpatialTree = new CBSPTreeLoader( m_pD3DDevice, m_bHardwareTnL,
                                             strFileName );

        m_pAlphaTree   = new CBSPNodeTree( m_pD3DDevice, m_bHardwareTnL );

        // Add our scene callback to the player.
        GetGameApp()->GetPlayer()->AddPlayerCallback( CScene::UpdatePlayer, this );

        // Attempt to load the file
        File.Load( strFileName );

        ...
        ...
```

This line is all that needs to be altered in order for our application to begin using the compiled BSP tree information generated and exported by the pre-processing tool developed in lab project 16.2. It is still also entirely possible for the application to select an alternate spatial partitioning tree class at any point, using the same compiled IWF file. The polygon data that is loaded by the scene will simply be compiled as has always been the case in earlier lab projects.

Lab Project Conclusion

This concludes our coverage of the first BSP tree rendering application. In the next lesson we will enhance both the compiler tool and the BSP tree loader class to make use of accurate visibility information. This will be used to cull away any level geometry that cannot possibly be seen due to any other geometry that might be occluding the player's view of other parts of the scene. This should greatly enhance both the performance of our rendering application, but will also open up many opportunities for improving the efficiency of many tasks that we must undertake in the future when developing additional game technologies.