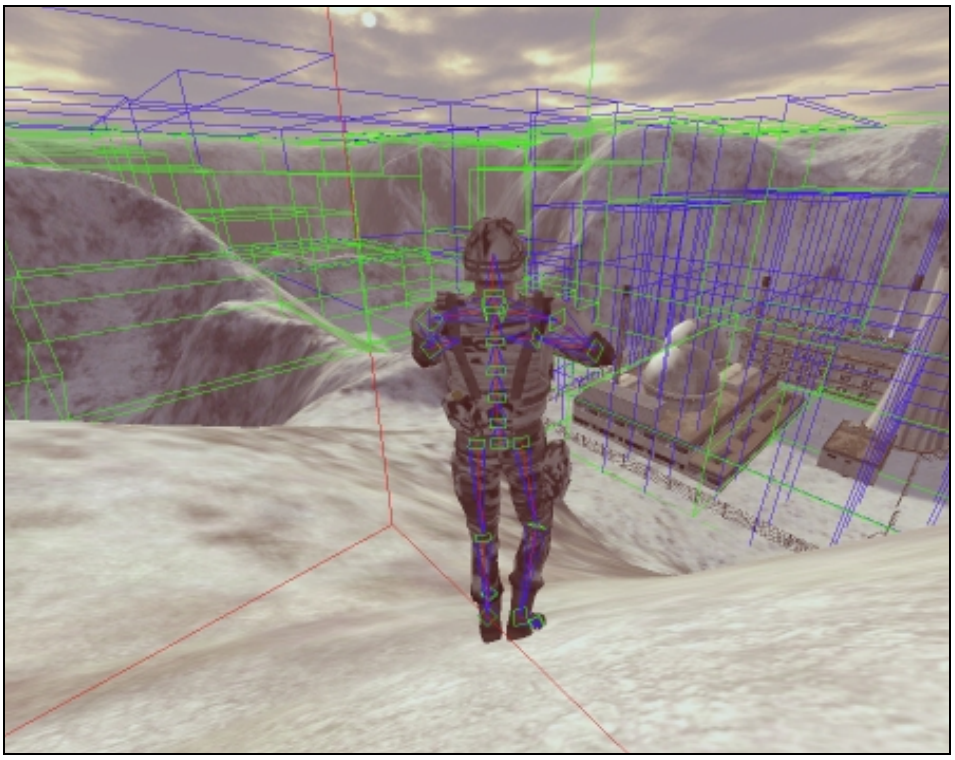
# Chapter Fifteen

## Hierarchical Spatial Partitioning II



# Introduction

In the previous lesson we developed various spatial trees for use with our collision system, allowing us to perform polygon queries on large scenes in a fraction of the time it would have otherwise taken. In this lesson we will add to the code that we developed and implement a hardware-friendly rendering system using the same spatial trees. Since our spatial tree will ultimately be responsible for rendering its static polygon data, we will provide a means for allowing the tree to render only the polygon data that is currently visible (i.e., sections of the scene that are contained or partially contained inside the camera frustum). From the application's perspective, rendering the tree will be similar to rendering a mesh. CBaseTree will expose a DrawSubset method (just like CActor and CTriMesh) which will instruct the tree to render all visible polygons that use a given attribute in a single call. This will allow the application to set the textures and materials used by a given subset before instructing the tree to render all currently visible polygons that use it. This will ensure that the static data in our tree can be batch rendered, minimizing the texture and material state changes that have to be performed by the application.

All of the tree types we studied have the same characteristics with regards to rendering. This will make CBaseTree an ideal location for the implementation of the render system. Thus we benefit from having only one implementation of the system that is shared by all derived types. All of our trees have leaves which are either currently visible or outside the frustum. They may or may not contain polygon data which will have to be rendered if the leaf is indeed considered visible. Theoretically, our rendering system has a simple job; traversing the tree and rendering only the polygons stored in visible leaves. However as trivial as this may seem, a naïve implementation can have drastic performance implications as we will discuss in the next section. We will have to develop a system that keeps the number of traversals to a minimum and introduces as little CPU processing overhead as possible when determining which data should be rendered. On the latest graphics cards, which are capable of rendering very large numbers of triangles, we could easily end up with a situation where rendering the entire scene brute force could far outperform our hierarchical frustum culled visibility system, if an inefficient system is put in place.

In the second part of this lesson we will add dynamic object support to our spatial trees. Although the application will be responsible for rendering its dynamic objects, by storing them in the spatial tree and having access to the leaves in which their bounding volumes exist, we can benefit from the visibility system such that the application need only render a dynamic object if it currently exists in a leaf that is flagged as visible. We will add methods to CBaseTree to insert a new dynamic object into the tree which will be called when the dynamic object is first created. The tree will have no knowledge of exactly what type of object has been assigned to it; it will view the concept of dynamic objects in a rather abstract way. This will ensure that our tree will not become dependent on certain object types such as CActor or CTriMesh. From the tree's perspective, each dynamic object will simply be an AABB and a context pointer which only has meaning to the application. Methods will also be added to the tree so that the application can update the current position of a dynamic object that is registered with the spatial tree. This will instruct the tree to remove the dynamic object's pointer from any leaves in which it is currently contained and use its AABB to find the new leaves its pointer should be added to. The tree will expose methods such as GetVisibleLeaves which will return a list of only the leaves in the tree which are currently visible. The application can then loop through these leaves and render only the dynamic

objects it finds stored there. Any dynamic objects assigned to invisible leaves will not be processed at all. For each dynamic object registered with the system the tree will maintain a list containing all the leaves that object is currently contained within. This allows the application to query the leaf list for a given dynamic object so that the application has knowledge of the leaves in which it is contained (whether those leaves are visible or not). In this section, we will also cover the application code that has been changed to support dynamic objects being used with an ISpatialTree derived class.

Let us begin our discussions by tying up the spatial tree's management of its static polygon data. In the previous chapter we discussed how to add, build, and run collision queries on this data. Now we must implement the final piece, the code that will render this static geometry using hierarchical frustum culling.

# 15.1 Rendering Spatial Trees

When we concluded our coverage of spatial trees in the previous chapter, each leaf stored the polygons it contained as an array of CPolygon pointers. This polygon data is located in system memory to make sure that it is easily accessible for collision queries. Consequently, it is not currently able to be efficiently rendered. We know that in order to get maximum performance from our rendering system we will want the geometry stored in vertex and index buffers located in video memory so that the GPU on the graphics hardware can access that data for transformation/lighting purposes as quickly as possible. However, we certainly would not want the only copy of the data our tree uses to be stored in vertex and index buffers since this would require the locking and reading these buffers during collision queries which would result in very inefficient collision tests. What we will need to do is create a second copy of the scene data that is stored in vertex and index buffers that can be optimally rendered by the hardware. This also means that we can perform optimizations on the renderable version of the data. For example, in this lesson a key optimization will be performing a weld to get rid of duplicate vertices. This is an important step since scenes often have separate vertices for each polygon, even when polygons share an edge and have the same attributes.

The classic example of why it is efficient to perform a weld on the render geometry can be seen if you were to place a cube in GILES™ and export it to an IWF file. For various reasons, GILES™ stores all of its data at the per-face level (even vertices), which means it will never share vertices between faces. We know that if we were to place a cube and assign the same attribute to each face of that cube, we would only need to have eight vertices that are all indexed by the six faces (twelve triangles) of that cube. However, GILES™ will actually export four separate vertices per face (quad) resulting in a cube consisting of 24 vertices, regardless of vertex location, material, UV coordinates, etc. Thus, in any given corner of the cube, there exists three vertices in the same location (one for each face that meets at that corner point). For rendering purposes, it is generally going to be preferable to have a shared pool of eight vertices that are indexed by all the faces of the cube. This is essentially what a weld does for us -- it finds these duplicated vertices and replaces them with a single vertex. All the faces that used those previous vertices then have their indices remapped to the new single vertex. The cube looks exactly the same, but we have just eliminated 2/3$^{rds}$ of our vertex data. If we imagine the savings that might be gained by welding an entire scene, we can see that this would allow us to fit much more geometry into a single vertex buffer and thus, minimize the number of vertex buffers we will need to store our scene

geometry. Remember that changing between vertex buffers can be an expensive operation, so we will want to do it as infrequently as possible. By minimizing the number of vertex buffers needed to store the scene, we minimize the number of vertex buffers we need to set and render from in a given frame.

There are many ways a rendering system can be implemented for a spatial tree. Indeed, quite a few designs were attempted during the development of this course. This was time consuming but fortunate as we discovered quite significant performance differences between the various systems. Even system designs that seemed quite clever and robust on paper performed terribly when compared against simple brute force rendering on high end machines with the latest graphics hardware. To be fair, a number of the maps and levels we tested had fairly low polygon counts (by today's standards), so we were not too surprised to see that they could be easily brute force rendered by the latest nVidia and ATI offerings. Also note that these scenes were not being rendered with complex shaders or multiple render passes, so the latest hardware could easily fill the frame buffer without any trouble. This was certainly true in tests where we used pre-lit scenes with a single texture.

Although this might seem an unfair test for any visibility system, the fact that brute force rendering was significantly outperforming these various system implementations on high end hardware, even when only half the scene was visible, is an important point that is really worth keeping in mind. It is a common mistake to assume that because the spatial visibility system might be rendering only half the scene when the player is in a given location within the game world, versus the brute force approach which is always rendering the entire scene, that the latter would always be slower. But it is critical to factor in the hardware in question. If the hardware is able to brute force render the entire scene without difficulty, the difference then becomes one of what is involved in the spatial system's determination of which polygons should be rendered. That is, we found in various spatial tree visibility systems that we implemented that the processing and collection time for visible data often exceeded the time it took to render the whole scene using brute force.

> **Note:** To be clear, we are talking solely about the rendering of the static scene polygon data here, which is perhaps being a bit unbalanced with respect to such spatial visibility schemes. One must also factor in the savings of quickly identifying and rendering dynamic objects which are only contained in visible leaves. A typical scenario might contain dozens of skinned characters for example, of which only two are contained in visible leaves. If these characters were each comprised of thousands of triangles, the savings over brute force would generally become more apparent. In this section though, we are focusing on trying to efficiently render the static scene in a way that helps rather than hinders performance on high end systems.

When the same systems were compared on much lower end machines, the situation was much better for hierarchies. That is, the extra CPU processing involved in collecting only the visible data made all the difference to the aging video hardware. Such cards could not render the entire scene brute force at interactive frame rates and the various visibility systems helped to achieve much higher frame rates. One might be tempted to conclude that the spatial tree's visibility system could be put in place to help our games run at acceptable speeds on lower end machines even if they provide no real benefit on the latest cutting edge systems (which seemed not to need much help). This is certainly a valid consideration, as it allows a game to run on a much wider variety of machines (which could mean the difference between decent sales and great sales for a commercial title). Certainly it is generally in a software company's best interest to implement systems that support a wide variety of target platforms as it opens up the potential market for the game. A recent survey showed that the vast majority of computer owners that play games

are still using nVidia geForce™ 2/3 (or equivalent) hardware, so there is certainly something to be said about that.

However, what was unacceptable about many of the system designs we tried was that while speeding up performance on low end machines, performance on high end machines was degraded (sometimes significantly so). Essentially, the time required to traverse the tree and collect the visible data became the main bottleneck. As mentioned, this situation would likely be quite different if we were rendering a huge scene with high polygon counts or a scene that required many rendering passes per–polygon to achieve advanced lighting and texturing effects. In such cases, even the latest graphics cards would find themselves hard pressed to perform brute force renders. This is where our spatial tree's visibility system could really make a difference as it only asks the hardware to render the geometry that is currently visible. This would once again allow for a reduction in the list of primitives to be rendered each frame which will hopefully represent an amount that can be easily rendered by the hardware.

We should also not neglect the far plane case when imagining how our spatial tree's visibility system can help out even powerful graphics cards. Imagine for example a game that implements a continuous terrain such as those found in such products as Microsoft Flight Simulator®. The terrains in such games would not only be too large to render brute force, but even on the latest hardware they might also be too large to even fit in memory at once. The terrain data would have to be streamed from disk as and when it comes into view and released when it is no longer visible. With our spatial tree's visibility system, any leaves that are outside the frustum could have their geometry data flushed from memory. With our hierarchical traversals, we could very easily detect which leaves are within the frustum each frame and stream the terrain data for that leaf from disk if it is not already loaded. We can then collect the data from each visible leaf and pass it to the pipeline to render. This task would be almost impossible to achieve without a spatial manager. It is for this reason that a spatial manager and visibility systems generally will be at the heart of nearly every commercial quality game title.

Although the benefits of a spatial tree for rendering purposes are numerous, there will be times when you will be rendering small and simple scenes which modern hardware can brute force render more efficiently than our spatial tree's visibility system. Although not using a spatial manager and visibility system in the days of the software engine would have been unthinkable, the latest graphics cards do their jobs very well and require less help from the application than in days gone by. As mentioned, when rendering simple scenes, trying to help the card instead of letting it just render everything can actually hinder performance in many cases. The visibility system we choose to implement then has to be one that prepares the visible data so efficiently that the time requirements are negligible in such situations. A classic example of why this is so important can be understood when we imagine the player located in the bottom left corner of the scene looking towards the top right corner of the scene with a wide FOV and a near infinite far plane. In such cases, the entire scene would most likely be visible, so the quickest solution would always be to perform an optimized brute force render of the entire scene, batched by subset. However, our visibility system would have to perform a tree traversal and collect the data at each leaf just to determine that the entire scene should ultimately be rendered anyway, just as in the brute force case. The issue is that our visibility system had to run some form of logic to ascertain this fact before it could start rendering anything. The brute force approach had no CPU processing to perform and could start rendering straight away. In such situations, the brute force approach is generally going to beat a basic frustum visibility system hands down. What we need to do is make sure that we implement a system that minimizes the damage to frame rate in these situations, but that when the player is

positioned at a location within the scene such that only a few leaves are visible (the general case), the visibility system will outperform the brute force system.

To sum up, we are going to try to design a system that significantly speeds up our rendering on both low and high end systems in the best case; in the worst case, where the scene could easily be brute force rendered by a high end graphics card, we want our implementation to speed up rendering on a low end machine but not under-perform brute force rendering on a high end system by any significant degree.

## 15.1.1 Different Rendering Schemes

There are many different schemes ranging from easy to difficult that can be employed to collect and render the visible data from a spatial tree. Unfortunately, the performance of such algorithms tends to vary from poor to good in that same order. In this section we will discuss some of the approaches that one might consider employing. We will discuss techniques that span the naïve to the complex and end with a discussion of the system that we ultimately ended up going implementing in CBaseTree.

### The UP Approach

The simplest way to render the visible the data is to draw the system memory polygon data stored at each visible leaf. DirectX has the IDirect3DDevice9::DrawPrimitiveUP method that allows us to render data directly using a system memory user pointer. With such an approach we do not need to make a copy of the polygon data in vertex or index buffers; we can pass a pointer to each CPolygon's vertex array directly into the DrawPrimitiveUP method. One approach would be to have an empty vertex array allocated for each subset used by the tree's static data prior to performing a visibility pass. These vertex arrays would be populated with the vertex data from each CPolygon from every currently visible leaf when the tree is traversed. After the visibility traversal is complete, this collection of vertices could be rendered. This array would then be flushed before the next visibility pass is performed.

As an example, if we imagine the static data in the tree contains polygons that use five different attributes in total, our tree would use five vertex arrays as bins in which to collect the vertex data from the visible leaves and add them to the array that matches their attribute ID. These arrays/bins would be emptied prior to the visibility process being performed each frame. We could implement a ProcessVisibility method that would be called by the application prior to rendering each subset of the tree. This method would traverse the tree, testing each node's bounding box against the camera's frustum so that any node that is currently outside the frustum will not have its children traversed into. Any child nodes and leaves underneath that node in the tree will not be traversed or processed and as such, any polygon data contained down that branch of the tree will be quickly and efficiently rejected from the collection process.

When a leaf is found that is visible, its visible Boolean will be set to true and a loop through each polygon stored there will be performed. For each polygon, we will check its attribute and copy its vertex data into the matching vertex bin. So if the current polygon we are processing in a leaf had an attribute

ID of 4, we would copy its vertices (one triangle at a time) into the 5th vertex array (VertexArray[4]). After the tree has been traversed, we would have built five vertex arrays and flagged each visible leaf. At this point, the ProcessVisibility method would return program flow back to the application.

The next stage for the application would be to loop through each subset and call the tree's DrawSubset method after setting the correct device states for that attribute. For example, the application would set the texture and material for subset 0 and call the tree's DrawSubset method, passing in this attribute ID. As the vertex array has already been compiled for all visible triangles that use this subset, this method can just call the IDirect3DDevice9::DrawPrimitiveUP method passing in VertexArray[0] and the number of triangles stored in this array. We would also state that we would like it rendered as a triangle list. Once the application had called DrawSubset for each attribute used by the tree, all visible triangles would have been rendered. By collecting the visible triangles into arrays first, we minimize the number of draw calls which would otherwise be in the tens of thousands if we rendered each polygon's vertices as and when it was located in a visible leaf during the visibility pass.

So we can see that using this approach, it is the ProcessVisibility method that actually traverses the tree and collects the triangle data for each subset. The DrawSubset method would simply render the vertex bin/array that matches the passed attribute that was compiled during the visibility pass.

**What is wrong with this approach?**

Although this method sounds delightfully easy to implement, it is a definite example of the wrong way to design this system. Performance would be simply terrible for a variety of reasons. First, as we traverse the tree, we have to perform a system memory copy of the vertex data from each CPolygon into the appropriate vertex bin. This would need to be done for every polygon currently visible. As vertex structures can often be pretty large, these memory copies really drain performance. When the entire scene is visible, we will be copying hundreds or thousands of vertices each frame and seriously tying up the CPU. The brute force approach in such situations would have none of this overhead and could simply just render the scene prepared in static vertex and index buffers with just five draw calls.

The second major flaw with this system is the use of the DrawPrimitiveUP method. Although it looks like the use of this function handily avoids the need to employ vertex buffers, the real concern is what happens behind the scenes. When we call the DrawPrimitiveUP method, a temporary vertex buffer will be created and locked and the vertex data we pass will then have to be copied into this vertex buffer (yet more copying of memory) before it is unlocked. After unlocking this vertex buffer, the pipeline may then have to commit this data to video memory, involving yet another copy of the vertex data over the bus into video memory. Transforming all the vertex data for the currently visible scene could mean a huge number of vertices will have to be passed over the bus each time.

As you can see, although this system it simple to implement, the performance would be quite poor. With an outrageous amount of memory copying to perform, this system is typically outperformed by brute forcing even on low end hardware. Memory copying is performance killer and in this case we are implementing three major copy stages. This is clearly not the way to go.

## The Dynamic Index Buffer Approach

The dynamic index buffer method is discussed in many published texts and as such is worth discussing here. With this approach we would invoke a process just after the tree is built which prepares the render data. It would allocate a static vertex buffer for maximum performance which will typically be allocated in video memory on T&L capable cards. We will also allocate a *dynamic* index buffer for each attribute used by the static scene stored in the tree. These index buffers will be flushed prior to each ProcessVisibility call.

The additional build process that is employed only once just after the tree is first built would loop through each leaf in the tree and copy the vertex data for each polygon into the static vertex buffer. As all the vertices of all polygons in all leaves will be stored in this vertex buffer we will also need to store the indices of each polygon in the leaves themselves. This is so we know for a given visible polygon, which section of the vertex buffer it is contained in and the sections that should be rendered or skipped. As each polygon is added to the vertex buffer, we will store the indices of its vertices in the leaf. After the post-build process is performed, we would have a single vertex buffer containing all the vertices of the static scene, and in each leaf we would have the indices for all the vertices of each polygon so that we know where they exist in the vertex buffer. We will also have an array of empty dynamic index buffers (one for each subset) which will be filled with the visible polygon indices during the ProcessVisibility call.

When the ProcessVisibility call is made by the application prior to rendering the tree's subsets, this function will first make sure that all the dynamic index buffers are empty. Then it would traverse the tree searching for visible leaves as in the previously described method. For each visible leaf we find, we would loop through each of its polygons and add its indices to the dynamic index buffer that is mapped to the attribute for that polygon. For example, if a given polygon had an attribute ID of 4, we would copy its indices into the fifth index buffer (indexbuffer[4]). When the ProcessVisibility method returns program flow back to the application, the tree would have the indices of all the visible triangles for a given attribute in each index buffer. The application would then bind the appropriate texture and material, set the appropriate states, and then call the tree's DrawSubset method to draw the required polygons. This method would simply call the IDirect3DDevice9::DrawIndexedPrimitive method for each dynamic index buffer it contains. We would pass DrawIndexedPrimitive the vertex buffer used by the entire scene and the index buffer of the current subset we are rendering. That index buffer would describe the indices of all the triangles in the vertex buffer that are visible for the current subset being rendered. After this has happened for each subset, all visible triangles will have been rendered. Thus, the number of DrawIndexedPrimitive calls would be equal to the number of subsets used by the polygons in the tree.

### What is wrong with this approach?

This design does perform much better than the previously discussed approach due to the fact that the vertex data was already contained in a video memory static vertex buffer. This meant that the sizable vertex structures did not have to undergo the multitude of copy phases. The vertex data was created just once at application startup and committed to video memory. However, on simple levels, brute force rendering significantly outperformed this technique on high-end hardware even when half the scene was being rejected in the visibility pass. This was especially true in the case of a clipped tree where the

polygon count was raised between 30% and 80% due to the polygon splitting performed in the build process. We can see for example that in the extreme case where there was an 80% increase in the polygon count, even when half the scene was being rejected, the polygon count had almost doubled and as such, half of our visible scene contained nearly as many polygons as brute force rendering the original polygon data. Therefore, brute force was still able to render the original scene faster than our clipped spatial tree could render only half of its split scene.

When using a non-clipped tree, things were better, but we still found that the memory copies performed at each visible leaf (polygon indices copied into the index buffer) are a bottleneck. Although index data is much smaller than vertex data and not nearly as expensive to copy or commit to video memory, when the entire scene was within the frustum, we were still dealing with tens (even hundreds) of thousands of indices to copy into the index buffer during each visibility determination pass. Combined with the fact that a dynamic index buffer typically performs a little slower than a static index buffer and we still had a system that, while much improved over the previous method, suffered poor performance compared to brute force when there was medium to high visibility in non-trivial, but not overly complex, scenes.

What was clear was that in order to get better performance we needed to eliminate the copying of vertex and index data and instead store the renderable data in static vertex and index buffers in the post build phase. This would allow our vertex and index data to reside in optimal memory using an optimal format for the driver. In the next section, we will discuss the rendering system we finally wound up implementing, which provided much better performance than the previous systems.

## 15.2 The CBaseTree Rendering System

The CBaseTree rendering system is not as complicated as it may first seem when looking at the code. Unfortunately, the code is made significantly less trivial by the fact that multiple vertex buffers may have to be used if the vertex count of the static scene exceeds the maximum vertex buffer size supported by the hardware. This necessitates the introduction of an additional level of structures that makes things a little difficult to follow at first. Because this reduces clarity of what is essentially a rather simple system, we will first discuss the system with this complication removed. That is, we will discuss this system assuming that we can use an infinitely large vertex buffer and as such, all the static geometry in the tree can always be contained within it. Once we understand the system, we will discuss how it needs to be adapted to cope with multiple vertex buffers.

The system operates in three stages much like the previous method we discussed. The first is executed only once when the tree is first built. It is this component that copies the vertex data of every leaf into a tree owned vertex buffer and the indices of every polygon into static index buffers (one for each subset). You will recall from our previous lesson that prior to the Build function of a derived class returning, it makes a call to the CBaseTree::PostBuild method to calculate the bounding boxes for CPolygons in the tree. This method also issues a call to a new CBaseTree member called BuildRenderData as shown below.

```
bool CBaseTree::PostBuild(  )
{
    // Calculate the polygon bounding boxes
    CalculatePolyBounds( );

    // Build the render data
    return BuildRenderData( );
}
```

It is the CBaseTree::BuildRenderData method that will build the static vertex buffer containing all the vertices of the tree's static geometry. It will also create an index buffer for each subset used by the tree and store the indices of each triangle in these buffers in leaf order. Let us discuss how our system will work and what processing this initial stage will have to perform.

This function will create a static vertex buffer to contain the vertex data and N static index buffers, where N is the number of attributes used by the static tree geometry. It will then loop through each leaf in the tree. For each leaf it is currently processing it will loop through all the CPolygon structures stored there. For each polygon, it will add its vertices to the vertex buffer and will test the attribute ID of the polygon so that its indices are generated and added to the correct index buffer. For example, if we are processing a polygon that uses attribute 4, we will add its vertices to the vertex buffer and then generate the indices for each triangle in the polygon based on the position it has been placed within the vertex buffer. Once we have the indices generated for each of its triangles we will add them to the fifth index buffer (index buffer 4) since this is the index buffer that will contain the indices of all triangles that use attribute 4.

The next bit is vitally important. When we add the indices of each polygon in the leaf to their respective index buffers, we will store (in the leaf) the first index where the triangles for this leaf start in each index buffer it uses. For example, if we are processing a leaf and we find that there are 6 triangles that use subset 2, we will add the indices of those six triangles to index buffer [1] and the leaf will be given the index where its indices begin in that index buffer and the number of primitives it contains for that leaf. As we process each leaf, we know for example that for a given subset that has polygons in leaf A, all of leaf A's indices will be added to that subset's associated index buffer in a continuous block. Going back to our previous example, if we find that a leaf has 6 triangles that use attribute 2 and the index buffer already contains 100 indices that were added when processing other leaves, the leaf will keep track of the fact that for subset 2, its polygons start at index 100 and the next 6 triangles from that point in this index buffer are its primitives. We will see why this is important in a moment.

Therefore, after this process is complete, if a given leaf contains polygons that use 4 subsets for example, that leaf will store an array of four RenderData structures (one for each subset it contains). Each RenderData structure contains the index at which this leaf's indices begin for a given subset in the associated index buffer and the number of triangles in that subset contained in the leaf. Thus, each RenderData structure stored in the leaf describes an index buffer range for a given subset. If the RenderData item stored in a leaf has an attribute ID of 5, it will describe a block of indices in the 6$^{th}$ index buffer (the static index buffer for triangles of subset 5) where this leaf's triangles start and end. We know then that if this leaf is visible, this section of that index buffer will need to be rendered.

We will also introduce a new class called CLeafBin. CLeafBin is really a wrapper class around an index buffer for a given subset containing methods to render its index buffer efficiently. As we need to run the collection of our render data as fast as possible, each RenderData item stored in a leaf will also store a pointer to the leaf bin for which it is associated. This way, during the traversal, when a visible leaf is found, for each RenderData structure stored (one for each subset contained in that leaf) in the leaf we can access the pointer to the associated leaf bin and inform it of a range in its index buffer that will need to be rendered.

Figure 15.1 shows a simplified diagram of the rendering system. In this example we will assume the tree contains only four leaves and uses only three attributes (the texture images describe the attributes). Study the image and then we will discuss its components.

In this example the tree contains only four leaves and these leaves can be seen in the top left corner of the image. Leaves 1 through 3 contain polygon data from two different subsets and leaf 4 contains polygon data from only one. The line dividing leaves 1 through 3 down their center is being used to indicate that these leaves have two RenderData structures because these leaves contain polygons from two different subsets. The fourth leaf contains polygon data from only the second subset and as such contains only one RenderData structure. At the bottom of the image we can see that as the tree's polygon data uses only three subsets, we allocate three leaf bins. Each leaf bin contains the static index buffer that will contain the indices of all the triangles in the tree that use that subset. Just like the tree's vertex buffer, these index buffers will also be populated just once after the tree has first been built. Do not worry about what the RenderBatch structures in each leaf bin are for at the moment; we will see how these will be used a little later to collect runs of visible triangles during the visibility determination process.

When the tree is first compiled, the leaf bin index buffers and the tree's vertex buffer will be empty. At this point, no leaves will contain RenderData structures. It is the job of the CBaseTree::BuildRenderData function to allocate these structures and populate the index buffers of each leaf bin and the vertex buffer of the spatial tree with data.

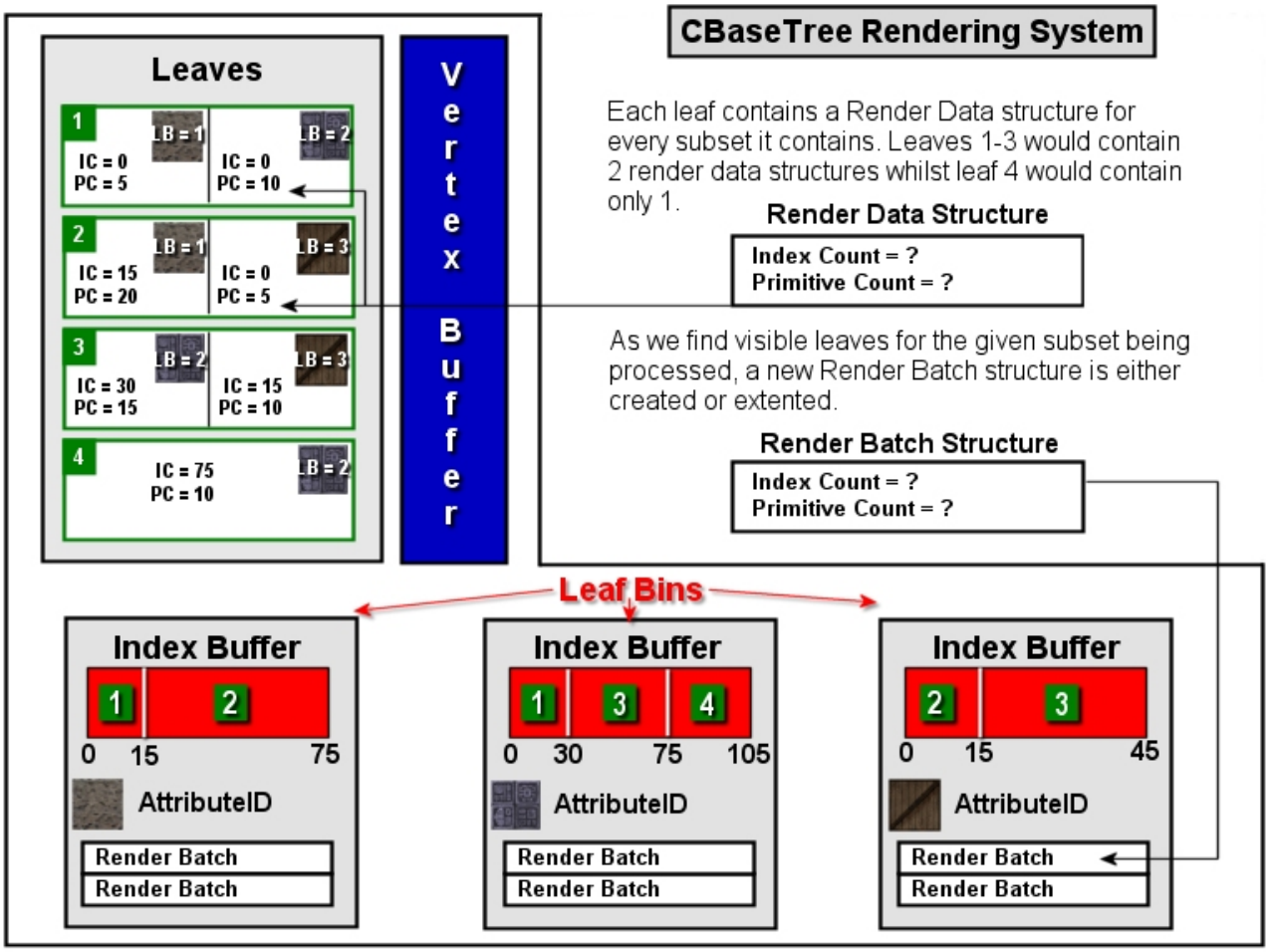



**Figure 15.1**

When the BuildRenderData function is first called, let us assume that it processes leaf 1 first. As it loops through each polygon, it finds that it contains 5 triangles that use subset 1 (the brick texture). It knows that it has to add the vertices of each of these triangles to the vertex buffer and the indices of each triangle (15 indices in total) to the index buffer stored in leaf bin 1 (the leaf bin associated with the first subset). It then notices as it adds these 15 indices to the index buffer that there are currently no indices in this index buffer. This means that this leaf's indices for subset 1 will be stored at the beginning of the index buffer contained in leaf bin 1. A new RenderData structure is allocated and added to the leaf which contains the subset ID of the leaf bin this data has been added to, the index into the index buffer of this leaf bin where the leaf's indices begin, and the number of triangles it has added to this index buffer. We can see if we look at the left half of leaf 1 in the diagram that its RenderData structure for leaf bin 1 contains an index count of 0, which describes where this leaf's subset 1 polygons begin in leaf bin 1 index buffer. It also contains a primitive count of 5. If we look at leaf bin 1, we can see that all 15 indices for the 5 triangles in leaf 1 have been stored at the beginning leaf 1's index buffer, exactly as described by the leaf's RenderData structure for this leaf bin.

Next we find that leaf 1 also contains 10 triangles which use subset 2. The vertices of each of these triangles are added to the vertex buffer and the indices of each vertex in the leaf that uses this subset are

generated and added to the index buffer of leaf bin 2. We are assuming in this example that these polygons are all triangles and as such this would add 30 indices to the index buffer in leaf bin 2. Once again, we create a new RenderData structure and store in it the leaf. This second RenderData structure will contain the leaf's information for leaf bin 2. We also notice that as this is the first time this subset has been encountered, there are currently no indices in the index buffer of leaf bin 2, so the RenderData structure for this leaf bin will store an index count of 0 and a primitive count of 10. At this point, leaf 1 has been processed. We know that it contains a RenderData structure for each subset it uses and each of these structures describes where its triangles exist in the associated leaf bin's index buffer. We can see for example that if this leaf is visible, when the application calls ISpatialTree::DrawSubset and passes in a subset ID of 1, we must make sure that the 5 triangles described by the first 15 indices in leaf bin 1's index buffer are rendered since these contain the triangles in leaf 1 for this subset. Likewise, when the application calls the DrawSubset method again to ask the tree to render subset 2, we know that if leaf 1 is visible, we must render the first 30 indices in leaf bin 2's index buffer.

We then move on to leaf two where we find 20 triangles that use subset 1 and 5 triangles that use subset 3. We add the vertices of each triangle to the vertex buffer and generate the indices for each triangle so that they correctly index into that vertex buffer. The 60 indices generated for the 20 triangles that belong to subset 1 are added to leaf bin 1's index buffer and a new RenderData structure is created and linked to this leaf bin via an attribute ID. Because there are already 15 indices in this index buffer, we know that the 60 indices for leaf 2's triangles must start at index 15 in leaf bin 1's index buffer. Therefore, we store an index count of 15 and a primitive count of 20 in this leaf's RenderData structure. This structure now tells us that the indices for the subset 1 triangles stored in this leaf begin at position 15 in leaf bin 1's index buffer and that there are 20 of them. We then process the other polygons in leaf 2 which in this example use subset 3. As this leaf bin has not yet had any data added to its index buffer, we know that it was empty before we added our indices for this leaf and therefore, a new RenderData structure is allocated and linked to this leaf bin via the subset ID. The index count of this structure is set to 0 as this leaf's indices for subset 3 polygons start at the beginning of this index buffer, and the primitive count is set to 5. The two RenderData structures stored in leaf 2 tell us that it contains polygons from two subsets, subsets 1 and 3. The indices of the triangles in this leaf that use subset 1 begin at index 15 in leaf bin 1's index buffer and there are 20 of them. We know that we can render this block of triangles if this leaf is visible. The second RenderData structure in leaf 2 tells us that this leaf also contains 5 triangles from subset 3 that start at position 0 in leaf bin 3's index buffer.

After this process has been repeated for all polygons in all leaves we will have the following:

- A static vertex buffer containing the vertices of all triangles in the tree.
- A Leaf Bin for every subset used by the tree's static data. Each leaf bin contains a static index buffer containing the indices of the tree's polygon data assigned to this subset.
- Each leaf will contain a RenderData structure for every subset its polygons use.
- Each RenderData structure stored in a leaf will describe where that leaf's indices start in its associated leaf bin and the number of triangles stored there (starting at that index).

If you refer back to Figure 15.1 you should be able to see how the index counts are calculated for each RenderData structure in each leaf. We can see for example leaf 4 contains only triangles that use subset 3. The indices of these triangles start at position 75 in the index buffer of leaf bin 2 and there are 10 triangles * 3 indices = 30 of them. The reason this leaf's indices start at 75 in leaf bin 2's index buffer is

that leaf 1 added 5*3=30 indices to this buffer first, followed by leaf 3 which added a further 15*3=45, taking the number of indices stored in this buffer up to 75 prior to leaf 4 being processed.

If you examine the contents of the three leaf bins in Figure 15.1 this should further clarify the relationship. At the end of the render data building process in this example, leaf bin 1 would contain 75 indices. The first 15 would describe the 5 triangles in leaf 1 which should be rendered if leaf 1 is deemed visible during the visibility processing pass. The indices from 15 to 75 describe the 20 triangles added to this index buffer from leaf 2. Therefore, we know that if leaf 2 is visible, indices 15 through 75 in leaf bin 1 should be rendered (when DrawSubset is called and passed a subset ID of 1). Furthermore, we can see that the RenderData structures in leaves 1 and 2 linked to this leaf bin describe these index ranges.

If we look at leaf bin 2 we can see that its index buffer contains triangles from three leaves and therefore, three leaves will contain RenderData structures linked to this leaf bin. Leaf 1 has added 30 indices to the beginning of this leaf bin to describe its 10 triangles that use subset 2. Leaf 3 also added 45 indices to this index buffer (starting at position 30) which describes its 15 triangles that use this subset. Finally, we can see that leaf 4 added a further 30 indices to this leaf bin describing the 10 triangles that it contains that belong to subset 2. Before continuing, study the diagram and make sure you understand the relationship between the leaf bins and the RenderData structures stored in each leaf.

The process described above is all performed by the CBaseTree::BuildRenderData method. This function is called once the tree is compiled and it builds and populates the RenderData structures stored at each leaf, the leaf bins for each subset, and the global vertex buffer that the index buffers in all leaf bins index into. The logical next question is, with this static data at our disposal, how do we efficiently collect and process it during the ISpatialTree::ProcessVisiblity call in a way that it can be efficiently rendered during a call to the ISpatialTree::DrawSubset method?

## 15.2.1 Render Batches

The heart of the rendering system lies in the leaf bin's ability to dynamically generate and store RenderBatch structures every time a visibility pass is performed on the tree (via a call to the ProcessVisibility function). A render batch essentially describes a list of triangles in a given leaf bin which are visible and stored consecutively in its index buffer. We can see in Figure 15.1 that render batch structures are generated and stored in the leaf bins (this takes place during the visibility pass). Each RenderBatch structure will describe a single call to the IDirect3DDevice9::DrawIndexedPrimitive method. Let us just take a look at that method to refresh our memory about its parameters. We will then be able to more easily see why a single RenderBatch structure represents a single call to this function.

```
HRESULT DrawIndexedPrimitive
(
    D3DPRIMITIVETYPE Type,
    INT BaseVertexIndex,
    UINT MinIndex,
    UINT NumVertices,
    UINT StartIndex,
    UINT PrimitiveCount
);
```

The important parameters to focus on, at least with respect to the current topic being discussed, are the last two. We pass in the location of the first index of a block of triangles we wish to render as the StartIndex parameter. The last parameter describes how many triangles (starting at StartIndex) we would like to render. Therefore, every time we call this method to render a series of triangles, the indices of those triangles must be consecutively ordered in the index buffer. For example, if we have an index buffer and we wish to render triangles 0 to 10 and triangles 40 to 50, we will have to issue two calls to this method, as shown below.

```
pDevice->DrawIndexedPrimitive( D3DPT_TRIANGLELIST,
                               0,
                               0,
                               NumOfVertices,
                               0,
                               10 );

pDevice->DrawIndexedPrimitive( D3DPT_TRIANGLELIST,
                               0,
                               0,
                               NumOfVertices,
                               120,            // 40*3
                               10 );
```

In the above example, NumOfVertices is assumed to contain the number of vertices in the vertex buffer bound to the device for these calls.

What is apparent by looking at the above example is that because our vertex and index buffers are static, when collecting the indices that need to be rendered from each visible leaf, we will not be able to render the triangles from multiple visible leaves with a single draw call unless these indices have been added to the index buffers of each leaf bin such that their indices follow on consecutively in the index buffer. Therefore the job of the leaf bins during a visibility pass will be to allocate RenderBatch structures to try to maximize the number of primitives that can be rendered in a single batch by detecting adjacent runs of visible triangles in the index buffer from different leaves. This will minimize the number of draw calls that will need to be made.

A RenderBatch structure is a very simple structure at its core that simply stores a start index and a primitive count as shown below. Essentially, each RenderBatch structure just represents a block of adjacent triangles in the index buffer that are currently considered visible and can be rendered via a single draw call.

```
struct RenderBatch
{
    ulong IndexCount;
    ulong PrimitiveCount;
};
```

To understand how it works, let us go back to the example tree illustrated in Figure 15.1 and assume for now that all four leaves are visible and that all the render data has been constructed as previously described. When the ProcessVisibility function is called by the application to determine leaf visibility,

the first thing it does is clear each leaf bin's RenderBatch list. That is because the job of the visibility pass will be to construct a list of render batch structures which can be executed to render the visible data.

The ProcessVisibility method will traverse the hierarchy searching for visible leaves. When a leaf is encountered which is visible, it will loop through each RenderData structure stored in that leaf. Remember, the number of RenderData structures stored in the leaf will be equal to the number of attributes/subsets used by the polygons in that leaf. Each RenderData structure would also store a pointer to the leaf bin to which it is assigned.

In the above example we would visit leaf 1 first and would determine that it is visible. We would then loop through the number of RenderData structures stored there, which in this example would be two. The first RenderData structure describes the index start and primitive count of all the triangles in leaf 1 that are stored in leaf bin 1. Therefore, we would call the leaf bin's AddVisibleData method, passing it the index count of the RenderData item and the number of primitives (shown in the following pseudo code). That is, we are informing the leaf bin of a run of visible triangles in its index buffer.

```
if (pLeaf->IsVisible)
{
   // Loop through each Render Data structure in this leaf
   for ( i = 0; i < pLeaf->RenderDataCount; i++  )
   {
       // Get current Render Data structure
       pRenderData = pLeaf->RenderData[i];

       // Fetch the leaf bin this Render Data structure is linked to
       pLeafBin = pRenderData.LeafBin;

       // Pass the index start location and primitive count into leaf bin
       pLeafBin->AddVisibleData( pRenderData->IndexCount ,
                                 pRenderData->PrimitiveCount );
   }
}
```

The above code essentially describes the visibility determination process. For each leaf we loop through the number of render data structures stored there. We fetch each one and retrieve a pointer to the leaf bin that this structure is defined for. We then pass in the index count and primitive count members of the render data item into the leaf bin so that it knows this block of triangles needs to be rendered.

Of course, it is the leaf bin's AddVisibleData method which is responsible for either creating a new render batch or adding the passed triangle run to a render batch that has been previously created. The latter can only be done if the value of IndexCount passed into the function carries on consecutively from the indices of the last visible leaf that was visited and added to that render batch. The following code snippet shows what the basic responsibility of this function will be.

In this code, certain member variables of CLeafBin are assumed to exist. There is assumed to be an un-initialized array of RenderBatch structures large enough to store the maximum number of render batches that could be created during a visibility pass for a given subset. This allows us to avoid the need to resize the RenderBatch array of the leaf bin every time it needs to have more elements added or emptied at the start of a new visibility pass. We simply reset the leaf bin's m_nBatchCount member to zero at the

beginning of each visibility pass and increment it each time a new render batch needs to be created. A new render batch only needs to be created if the block of triangles described by the input parameters to the function does not follow on from those stored in the current render batch being compiled. Look at the code first and then we will discuss its basic operation. Understanding this logic is vitally important as it is the key collection process in our rendering system. It is our means of rendering as many consecutively arranged triangles in a single draw call as possible.

```
void CLeafBin::AddVisibleData( unsigned long IndexStart,
                               unsigned long PrimitiveCount )
{

    ULONG LastIndexStart      = m_nLastIndexStart,
          LastPrimitiveCount  = m_nLastPrimitiveCount;

    // Build up batch lists
    if ( LastPrimitiveCount == 0 )
    {
        // We don't have any data yet so just store initial values
        LastIndexStart     = IndexStart;
        LastPrimitiveCount = PrimitiveCount;

    } // End if no data
    else
    if ( IndexStart == (LastIndexStart + (LastPrimitiveCount * 3)) )
    {
        // The specified primitives were in consecutive order,
        // so grow the primitive count of the current render batch
        LastPrimitiveCount += PrimitiveCount;

    } // End if consecutive primitives
    else
    {
        // Store any previous data for rendering
        m_pRenderBatches[pData->m_nBatchCount].IndexStart     = LastIndexStart;
        m_pRenderBatches[pData->m_nBatchCount].PrimitiveCount = LastPrimitiveCount;
        m_nBatchCount++;

        // Start the new list
        LastIndexStart     = IndexStart;
        LastPrimitiveCount = PrimitiveCount;
    } // End if new batch

    // Store the updated values
    m_nLastIndexStart     = LastIndexStart;
    m_nLastPrimitiveCount = LastPrimitiveCount;
}
```

The leaf bin object is assumed to have member variables that are used temporarily by the visibility pass to track the last index start and primitive count passed into the function. These will be set to zero at the start of the visibility determination process for a given frame. At the end of the function the parameters passed in are stored in these member variables so that they are accessible to us when the next visible leaf is encountered and this method is called again. These variables represent our means of determining whether the index start parameter passed in describes the start of a run of triangles that follows on

consecutively from the data that was added to the render batch in the previous call. If so, these triangles can be added to the current render batch simply by incrementing the primitive count of the render batch currently being compiled. Otherwise, the new triangles we wish to add do not follow on consecutively in the leaf bin's index buffer, so we will need to end our current batch and start compiling a new render batch. Let us analyze the above function a few lines at a time so that we are sure we really understand how it works.

This function is called from a visible leaf when processing its render data structures. For example, if the RenderData structure for a visible leaf is linked to leaf bin 1, then the index count and primitive count stored in that structure are passed into the AddVisibleData method of the relevant leaf bin during the visibility pass for the tree. We are essentially informing the leaf bin that the current leaf has triangle data which is relevant to the leaf bin's attribute ID and should therefore be rendered.

```
void CLeafBin::AddVisibleData( unsigned long IndexStart,
                               unsigned long PrimitiveCount )
{

    ULONG LastIndexStart      = m_nLastIndexStart,
          LastPrimitiveCount  = m_nLastPrimitiveCount;
```

The IndexStart parameter will contain the position in the leaf bin's index buffer where this leaf's triangles begin and the primitive count will describe the number of triangles (starting from the IndexStart parameter) in the leaf bin's index buffer that belong to the calling leaf.

The first thing we do in the above code is fetch the LastIndexStart and LastPrimitiveCount members into local variables. These will both be set to zero at the start of the visibility pass and as such, will be zero when this function is first called when the first visible leaf is encountered for a given leaf bin. If these are zero, then it means we have not yet created any render batch structures in this visibility pass and this is the first time the function has been called. If this is the case, we simply store the passed index start and primitive count in the LastIndexStart and LastPrimitiveCount members. We are essentially starting a new batch collection process. At the bottom of this function we will then store these values in the m_nLastIndexCount and the m_nLastPrimitiveCount members so that when the next leaf which is found to be visible sends its data into this function, we have access to the information about the batch we are currently compiling. No other action is taken if this is the first time this function has been called for a given visibility pass, as shown in the following conditional code block.

```
    // Build up batch lists
    if ( LastPrimitiveCount == 0 )
    {
        // We don't have any data yet so just store initial values
        LastIndexStart     = IndexStart;
        LastPrimitiveCount = PrimitiveCount;
    } // End if no data
```

However, if LastPrimitiveCount does not equal zero then it means this is not the first time this method has been called for this leaf bin during the current visibility pass. In other words, this is not the first visible leaf that has been encountered that contains triangle data for this subset. If this is the case, then LastIndexStart will contain the location of the index that starts a consecutive run of triangles we are

currently collecting and LastPrimitiveCount will contain the number of triangles we have collected in the adjacent run so far. If the index start that has been passed into the function is equal to that of the start of the current batch we are building (LastIndexStart) plus the number of adjacent indices we have collected for this run so far (LastPrimitiveCount), it means the triangles of the leaf that called this method are arranged in the leaf bin's index buffer such that they continue the adjacent run of triangles we are currently trying to collect (i.e., the current batch we are compiling). When this is the case, we can simply increase the current primitive count for the current batch.

```
    else
    if ( IndexStart == (LastIndexStart + (LastPrimitiveCount * 3)) )
    {
        // The specified primitives were in consecutive order,
        // so grow the primitive count of the current render batch
        LastPrimitiveCount += PrimitiveCount;

    } // End if consecutive primitives
```

Finally, if none of the above cases are true it means that the index start passed in describes the location of a run of triangles for the leaf that does not exist immediately after the run of triangles we have compiled for the current batch. Thus, we have reached the end of a run of triangles that can be rendered in one batch. At this point, LastIndexStart and LastPrimitiveCount (with their values set in the previous call) will contain the start index and primitive count of a block of adjacent triangles that were being collected up until this call and therefore describe a block of triangles that can be rendered. We can add no more triangles to this batch. The current leaf data that was passed into this function cannot be added to this batch, so the previous index start and the primitive count collected so far are stored in a new RenderBatch structure, added to the leaf bin's RenderBatch array, and the number of render batches is increased. The LastIndexStart and LastPrimitiveCount members are then set to those values that were passed into the function, thus beginning a new cycle of trying to collect adjacent runs for a new batch. The remainder of the logic is shown below.

```
    else
    {
        // Store any previous data for rendering
        m_pRenderBatches[pData->m_nBatchCount].IndexStart     = LastIndexStart;
        m_pRenderBatches[pData->m_nBatchCount].PrimitiveCount = LastPrimitiveCount;
        m_nBatchCount++;

        // Start the new list
        LastIndexStart     = IndexStart;
        LastPrimitiveCount = PrimitiveCount;
    } // End if new batch

    // Store the updated values
    m_nLastIndexStart     = LastIndexStart;
    m_nLastPrimitiveCount = LastPrimitiveCount;
}
```

Understanding this process of collecting render batches is easier if we use some examples. Figure 15.2 shows the example arrangement we discussed earlier. We will now discuss the visibility processing step

performed when ProcessVisibility is called. In this example we will assume that all leaves except leaf 3 are visible. Just remember that no render batches exist in any leaf bins at the start of a visibility pass.

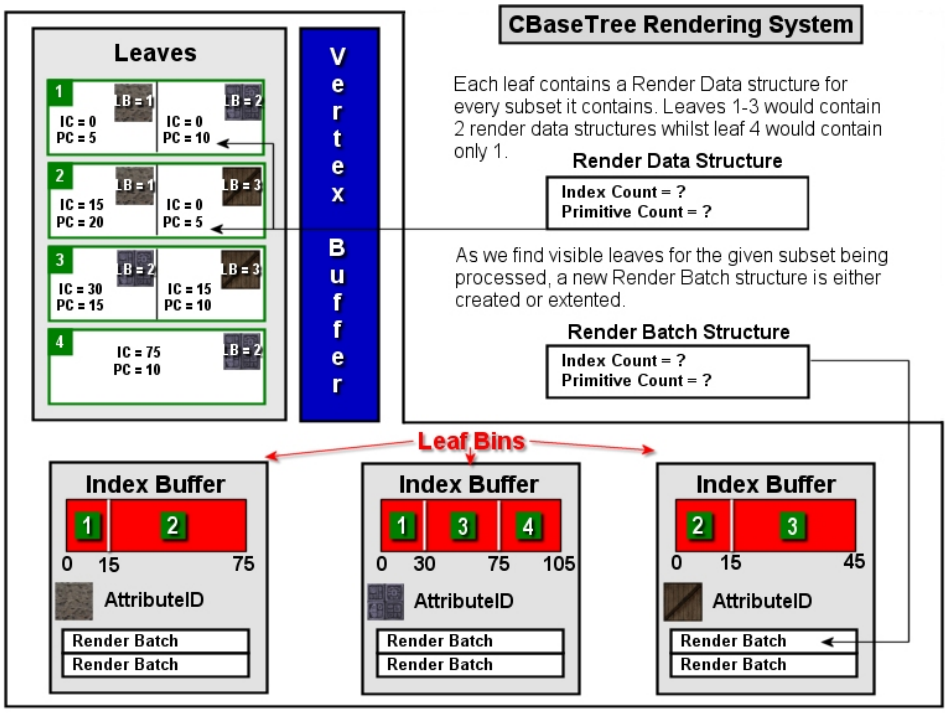



**Figure 15.2**

Leaf 1 is visited first and its first render data structure is processed. As this describes triangles that are contained in leaf bin 1, this leaf bin's AddVisibleData method is called and passed an index start of 0 and a primitive count of 5. As leaf bin 1 has not yet had this method called for any other leaf during the current visibility pass, we simply store 0 and 5 in its LastIndexStart and LastPrimitiveCount members. We then process the second render data structure in this leaf for leaf bin 2 and call its AddVisibleData method as well. An index start value of 0 is passed and a primitive count of 10 is passed. Since this is the first time leaf bin 2's AddVisibleData method has been called, the same thing happens. We simply store the passed index start and primitive count values in the LastIndexStart and LastPrimitiveCount members. At this point, no render batches have been created in any leaf bin, but in leaf bins 1 and 2, we have started compiling a batch.

Next we find that leaf 2 is visible, so we process its two render data structures. The first one is for leaf bin 1 again, so its AddVisibleData method is called and passed an index start of 15 and a primitive count of 20. However, this time when the method is called we detect that the index start passed in 15, is equal to the LastIndexStart value recorded (0) plus the last primitive count recorded (5) multiplied by 3. Therefore, we know that the triangles of leaf 2 stored in this leaf bin's index buffer follow on exactly from the triangles of leaf 1 that were passed in the previous call to this function. In this case, we can simply leave the LastIndexStart value of the leaf bin at its previous value (0) (which describes the starting location for the batch we are compiling) and increase the LastPrimitiveCount by the primitive count passed in. The value of LastPrimitiveCount is therefore increased from 5 (leaf 1's triangles) to 25 (adding leaf 2's 20 triangles). The current batch we are compiling now describes 25 triangles that can be rendered with a single draw call.

At this point in the discussion we will skip ahead to the end of the process as we can see that no further visible leaves exist that contain triangle data for this leaf bin. Therefore, at the end of the process we will simply create a single render batch for leaf bin 1 that has an index start of 0 and a primitive count of 25.

This render batch can be rendered with a single DrawIndexedPrimitive call and we can see that we have been able to merge the triangles from leaf 1 and leaf 2 that use this subset into a single render batch. Later, when the application calls ISpatialTree::DrawSubset and passes in the subset ID assigned to the first leaf bin, we can simply extract the index start and primitive count values from the leaf bin's only render batch and pass them as parameters to the DrawIndexedPrimitive call. This system of collecting adjacent runs into render batches allows us to lower the number of draw calls that will have to be made to render the triangles of a single leaf bin by consolidating adjacent runs into render batch instructions. Let us now continue with our example so that we can see where and why multiple render batches will sometimes need to be created for a given leaf bin.

When we left off in our example, we had just processed the first render data structure in leaf 2. When we process the second render data structure in this leaf (for leaf bin 3) we call leaf bin 3's AddVisibleData method and pass in an index start of 0 and a primitive count of 5. As this is the first leaf we have encountered that has data for leaf bin 3, this method simply copies the values of the past index start and primitive count parameters into leaf bin 3's LastIndexStart and LastPrimitiveCount members, thus starting the compilation of a new render batch..

Now we get to leaf 3 and determine that it is outside the frustum and therefore not visible. So we can skip it and its triangle data is never added to any leaf bins. We can once again skip ahead to the end of the process and see that in this instance, leaf bin 3 will have a single render batch generated for it with an index start of 0 and a primitive count of 5.

Finally, we get to the next visible leaf (leaf 4). This leaf contains 10 triangles that are stored in leaf bin 2 starting at location 75 in the index buffer. Therefore, we pass the values of 75 and 10 into leaf bin 4's AddVisibleData function. Now for the important part!

Leaf bin 2 has already been called during this visibility pass when leaf 1 was processed. Leaf 1 passed in a primitive count of 10 and a starting index 0. Therefore, the LastIndexStart member in this leaf bin will currently hold a value of 0 and the LastPrimitiveCount member will currently hold 10. We can see just by looking at the index buffer for leaf bin 2 in Figure 15.2 that leaf 4's triangles do not follow on from leaf 1's triangles. Leaf 3's triangles are inserted between them, but these triangles are not to be rendered because leaf 3 is not visible. In the code we just showed, we tested if the passed index start value described a consecutive run by testing to see if multiplying LastPrimitiveCount by 3 and adding it to LastIndexStart was equal to the index start value passed in. If they are the same, then the triangles we pass in continue on from those last added to the current batch we are compiling. Let us just perform a dry run of the function code shown above to make sure that this is definitely not considered an adjacent run:

```
if ( IndexStart == LastIndexStart + (LastPrimitiveCount * 3) ) )
{
  Increase LastPrimitiveCount
}
```

When this function is called by leaf 4, a value of 75 will be passed as the IndexStart parameter since this is where this leaf's triangles start in the index buffer for leaf bin 2. Furthermore, LastIndexStart will currently be set to 0 and LastPrimitiveCount will currently be set to 10 (leaf 1's primitive count) so this equates to the following test:

```
if ( 75 == 0 + (5 * 3)) )
{
  Increase LastPrimitiveCount
}
```

This equates to:-

```
if ( 75 == 15 )
{
  Increase LastPrimitiveCount
}
```

Clearly we can see that the above condition is not true and the triangles in leaf 4 cannot be rendered in the same draw call as those already collected from leaf 1. Therefore, the current value of LastIndexStart and LastPrimitiveCount will be stored as a new render batch in leaf bin 2 and LastIndexStart and LastPrimitiveCount will be set to the values stored in leaf 4's render data structure. When the process is over, these will also be added as a new render batch so that when visibility processing is complete, leaf bin 1 will have one render batch which renders the triangles from leaves 1 and 2 in a single draw call. Leaf bin 2 will have two render batches. One will render the triangles from leaf 1 and the other will render the triangles from leaf 4. This will have to be done with two separate draw calls. Leaf bin 3 will also contain a single render batch describing the triangles from leaf 2 that are visible and can be rendered with a single draw call.

Although this is a simple example, it is easy to imagine that during the tree traversal we will find many leaves that are visible and that have their triangle data arranged consecutively in the index buffers of the leaf bins to which they pertain. When such cases occur, we can render the triangles from multiple leaves with a single draw call instead of having to issue a DrawIndexedPrimitive call for each visible leaf that contains renderable data.

One thing that is immediately obvious about this system is that its success is dependant on the order in which we add the indices to the index buffer of each leaf bin during the BuildRenderData method. We can see in Figure 15.2 for example that because we process each leaf in the tree and add their indices to each leaf bin in the same order that we traversed those leaves during the visibility pass, we maximize the chance of collecting adjacent runs of triangles in multiple leaves. If we added leaf 2 to the index buffer first, followed by leaves 4, 3 and 1 (in that order) but still traversed the tree during the visibility pass in the 1,2,3,4 leaf order, we would get no adjacent runs and would end up having to perform a draw call for each leaf. That is, the number of render batches stored in each leaf bin would be equal to the number of leaves that contain polygons that use that leaf bin's attribute. This could be a very costly number of draw calls if every leaf was visible.

Luckily, we know that if every leaf is visible then our traversal methods will traverse that tree in the same order every time. Therefore, in the BuildRenderData method, one of the first things we will do is feed the entire scene's bounding box into the CollectLeavesAABB method. As we have passed the scene's bounding box to this function which encompasses all child leaves, the entire tree will be traversed and a list of all the leaves returned. However, the CollectLeavesAABB method would have added the leaves to the passed leaf list in the order in which the tree is traversed. That is, if the entire scene is visible during the ProcessVisibility call, this is the exact order in which the leaves will be

visited. Therefore, we will populate the leaf bins in that order by simply iterating through the leaf list returned from CollectLeavesAABB so that the indices of a leaf in the index buffer follow the indices of the previous leaf that will be visited during the traversal. In the case where the entire scene is visible, we will be adding visible data to each leaf bin's render batch system in the exact order that they are stored in the leaf bin's index buffer during the build phase. In this scenario, each leaf bin would create only one render batch that describes all the triangles stored in its index buffer. This means in the case where everything is visible, the number of draw calls will be reduced to the number of subsets/leaf bins in use by the tree.

Of course, when only some of the tree is visible there will be entire branches of the tree that are rejected from being further traversed and therefore, this will essentially end the render batch currently being compiled for each leaf. This is because the branch of the tree we have just rejected will describe a large section of the index buffer that we do not want to render and therefore ends any block of adjacent triangles we are trying to find. However, we at least know that if an entire branch of the tree is visible, because the visibility traversal order is the same order in which we added to the leaves to the index buffer of each leaf bin, all the leaves down that visible branch will be contained in the leaf bins as a continuous block of indices that can be represented as a single render batch and rendered with a single draw call.

## 15.2.2 Rendering the Tree

After the CBaseTree::ProcessVisibility call has been made by the application for a given frame update, the render batches for each leaf bin will have been constructed and will contain the triangles that need to be rendered for each leaf bin/subset. The application renders a given subset of the tree by calling the ISpatialTree::DrawSubset method. The code to such a function is shown below. Basically, it passes the drawing request on to the leaf bin's Render function. This is not the actual code from our lab project and all error checking has been removed. We only introduce it at this time to provide some insight into how the code will work. In the following function, the GetLeafBin method is exposed by CBaseTree and fetches the pointer to the leaf bin for the passed subset ID.

```
void CBaseTree::DrawSubset( unsigned long nAttribID )
{
    // Retrieve the applicable leaf bin for this attribute
    CLeafBin * pLeafBin = GetLeafBin( nAttribID );

    // Render the leaf bin
    pLeafBin->Render( m_pD3DDevice );
}
```

As the tree stores a leaf bin for each subset, this function simply fetches the leaf bin from its leaf bin array for the passed subset ID (attribute ID) and then issues a call to its Render function. Since the leaf bin has already compiled the render batches during the ProcessVisibility call, all it has to do is loop through each currently stored render batch and call the DrawIndexedPrimitive function to render the adjacent triangles described by each batch.

The code to such a function is shown below. Once again, this is not the actual code we will be using and it has had some error checking removed, but it does provide insight into the simple task assigned to the CLeafBin::Render method.

```
void CLeafBin::Render( LPDIRECT3DDEVICE9 pDevice )
{
    ULONG j;

    // Bin leaf bins index buffer to the device
    pDevice->SetIndices( pData->m_pIndexBuffer );

    // Render the leaves
    for ( j = 0; j < m_nBatchCount; ++j )
    {
          RenderBatch &Batch = m_pRenderBatches[j];

          // Render any data
          pDevice->DrawIndexedPrimitive( D3DPT_TRIANGLELIST,
                                         0,
                                         0,
                                         m_nVertexCount,
                                         Batch.IndexStart,
                                         Batch.PrimitiveCount );
    } // Next Batch
}
```

As you can see, this function is delightfully short and easy to follow. When the application requests that the tree render one of its subsets, we can see that it is really asking one of the leaf bins to render all of its visible triangles. The function first binds the leaf bin's index buffer to the device and then loops through each RenderBatch structure compiled during the ProcessVisibility pass for that leaf bin. For each render batch, it calls the DrawIndexedPrimitive method of the device passing in the start index and primitive count stored in the current render batch being processed. This describes the block of adjacent triangles in the index buffer represented by that render batch. In this example we will assume that m_nVertexCount is a member variable of CLeafBin and that it contains the number of vertices in the tree vertex buffer (i.e., the vertex buffer indexed into by all leaf bins).

Hopefully, you now see that our basic system is fairly logical and easy to understand. It is also fast because it allows us to work with static vertex and index buffers. Of course, this comes at the expense of potentially many more draw calls having to be issued than in the dynamic index buffer method, but it eliminates the abundance of memory copies that had to be performed to fill the dynamic index buffers during the visibility traversal. This proved to be more than a worthwhile tradeoff as our render batch system far outperformed the dynamic buffer approach in every test we ran.

## 15.2.3 Multiple Vertex Buffers

Thus far, we have been able to keep our system very simple with only a few structures needed to accomplish our goal. Essentially, we have leaves that contain information which describe their run of adjacent triangles in each leaf bin's index buffer, we have a leaf bin for each subset in use by the tree,

and we have RenderBatch structures that are built and collected in each leaf bin during a visibility pass. Finally, rendering a subset of the tree simply means asking the leaf bin associated with that subset to execute all its render batch instructions that were generated during the last visibility pass. Up until now we have assumed that all the vertices of all the polygons compiled into the tree can be stored in a single vertex buffer which is owned by the tree and indexed into by the single index buffer stored in each leaf bin. Unfortunately, vertex buffers cannot be of infinite size and furthermore, the maximum size of a vertex buffer which we can index into depends on the capabilities of the 3D device on which the application is running. One graphics card might be perfectly happy with using indices that reference vertices in a single buffer up to and beyond 16,000,000 (for example), while another card may insist on using vertex buffers containing less that 125,000. At the end of the day, we will need to make sure that the number of vertices we store in any vertex buffer is not greater than the maximum vertex index supported by the card. Therefore, if we find for example that we have 128,000 vertices comprising the static geometry of our spatial tree, but the system on which the application is currently running supports a maximum vertex buffer size of 64,000, we will have to store our 128,000 vertices in two vertex buffers (each holding 64,000). Each smaller vertex buffer is within the scope of the device's capabilities and as such, when we render the tree, we will have to render each vertex buffer (and their associated index buffers) separately.

This obviously has some ramifications for our system and forces us to add an additional layer of structures that we would rather not have to add. However, as we have discussed the system in the first part of this lesson without the multiple vertex buffer complication, adding it to our discussion now and discussing the changes that will have to be made will be much simpler to follow.

Because our tree may be using multiple vertex buffers to store its static geometry, we will have to make some fairly serious modifications to our system. First, during the BuildRenderData phase we will have to detect when the vertex buffer we are currently adding vertices to is full (i.e., it exceeds the maximum vertex indexing capabilities of the device) and create a new vertex buffer. To keep things simple, we will also create new index buffers for each leaf bin when this happens so that in each bin, we have separate index buffers for every vertex buffer being used by the tree. That is, if the tree is using five vertex buffers to store its geometry and a particular leaf bin contains triangles from each of these five vertex buffers, it will have five index buffers generated for it. So we will keep a logical pairing between vertex buffers and index buffers. We know when rendering the leaf bin that if it has five index buffers, we will have to loop through each, set the corresponding vertex and index buffers, and then render the triangles in that index buffer.

We will also need to know which vertex buffer a given leaf's RenderData structure is relative to. For example, we know that the RenderData structure in a leaf describes the index start and primitive count of one of its subsets. If its AttributeID is 2, this means it describes triangles in leaf bin 2. The problem is, leaf bin 2 may now have multiple index buffers, so we will need to store which index buffer / vertex buffer combination should be bound to the device in order to render the batch of triangles stored at that leaf. For example, we might imagine a case where leaf 1 has a RenderData structure that describes triangles in vertex buffer 1 and index buffer 1, but leaf 100's RenderData structure could contain exactly the same index start and primitive count values for the exact same subset but describe a run of adjacent triangles in the leaf bin's second vertex / index buffer pair. So we can see that each leaf's RenderData structure will have to store the numerical index of the vertex/index buffer pair that contains the vertices

of the triangles it describes. We might imagine at this stage then that the RenderData structure would look something like this:

```
struct RenderData
{
   unsigned long IndexCount;
   unsigned long PrimitiveCount
   unsigned long VBIndex;
   unsigned long AttributeID;
   CLeafBin     * pLeafBin;
}
```

Looking at this structure we can see that it contains not only the start index of where its indices start in the associated leaf bin's index buffer and the number of triangles stored there, but also contains the numerical index of the vertex buffer/index buffer combination that store the triangle data. We can see in this example how it might also contain the attribute ID which describes the subset/leaf bin this RenderData structure is associated with and a pointer to the leaf bin that stores triangles for this subset.

Storing the leaf bin pointer in the RenderData structure is very useful for speeding up the visibility process. When processing the RenderData structure for a given visible leaf, we have to pass its index count and primitive count into the leaf bin's AddVisibleData method so that it can be added to an existing render batch or used to start a new batch. By having this pointer stored in the data structure, we negate the need to search for the leaf bin with a matching attribute ID. We really need to make the tree traversal and visible polygon collection process as quick as possible, so we do not want to waste time looping through the leaf bin array finding the leaf bin the triangles in this RenderData structure should be added to. Therefore, we will find this during the building process and store the leaf bin's pointer in the RenderData structure for immediate access.

We can now see that the RenderData structure not only describes the number of triangles in a leaf bin and their location within the leaf bin's index buffer, it also describes to the leaf bin which vertex buffer should be bound to the device and which of the leaf bin's index buffers the triangle run is contained in. For example, if VBIndex were set to 3, this would inform the leaf bin that the vertices of these triangles are stored in the third vertex buffer in the tree's vertex buffer array and that its indices are stored in the leaf bin's third index buffer.

Unfortunately the provisions made here for the multiple vertex buffer case do not cover all scenarios we may encounter. For example, imagine the following situation during the BuildRenderData method:

We are currently visiting leaf 10 and we are processing the polygons in that leaf that use subset 6. Assume that there are 20 polygons in this leaf that use subset 6 and that up until this point we have been adding the vertex data to a single vertex buffer. Now imagine that as we add the first 10 triangles of this leaf's subset 6 polygons to the vertex buffer (and its indices to leaf bin 6's index buffer) we discover that we have filled up the vertex buffer. We have encountered a situation where we have to switch to a new vertex buffer (and create a new index buffer to go with it in leaf bin 6) halfway though processing the polygons for a single subset. Although this might happen only rarely, it is certainly possible that triangles from the same leaf that share the same subset are split over multiple buffers. In such a case, the RenderData structure should describe not only the vertex and index buffer to use for a given subset in

that leaf, but should describe all the vertex buffers/index buffers that contain subset 6 in that leaf. If we take this into account, we end up with a RenderData structure stored at each leaf that looks like so:

```
struct RenderElement
{
  unsigned long IndexStart;
  unsigned long PrimitiveCount
  unsigned long VBIndex;
};


struct RenderData
{
   RenderElement *pRenderElements;
   unsigned long ElementCount;
   unsigned long AttributeID;
   CLeafBin     * pLeafBin;
}
```

Let us stop and think about what information we are actually storing here. The RenderElement structure now stores the index count and primitive count for a single vertex/index buffer combination for a given subset. In the above example where the 20 polygons that used subset 6 in a leaf were spread over two vertex/index buffers, the RenderData structure in that leaf for subset 6 would contain two render elements in its array and would describe two runs of triangles in two of the leaf bin's index buffers. The VBIndex of the first would be 0 describing the first vertex buffer and the VBIndex of the second would be 1. When we reach a visible leaf during the visibility process, the leaf bin would now need to be passed the VBIndex of the render element alongside the index start index and primitive count of each element stored there. This is because, now that the leaf bin maintains an array of index buffers (one for each primary vertex buffer being used by the system), an array of RenderBatch lists will now need to be stored also. That is, the leaf bin will need to compile a render batch list for each vertex/index buffer pair in use by the leaf bin.

The following code shows what happens when a visible leaf is encountered during the ProcessVisibility call and the leaf's SetVisible function is called.

```
void CBaseLeaf::SetVisible( bool bVisible )
{
    ULONG                 i, j;
    RenderElement       * pElement;
    CLeafBin            * pLeafBin;
    RenderData          * pData;

    // Flag this as visible
    m_bVisible = bVisible;

    // If we're being marked as visible, inform the renderer
    if ( m_bVisible && m_nRenderDataCount > 0 )
    {
        // Loop through each renderable set in this leaf.
        for ( i = 0; i < m_nRenderDataCount; ++i )
        {
```

```
            pData    = &m_pRenderData[i];
            pLeafBin = pData->pLeafBin;

            // Loop through each element to render
            for ( j = 0; j < pData->ElementCount; ++j )
            {
                pElement = &pData->pElements[j];
                if ( pElement->PrimitiveCount == 0 ) continue;

                // Add this to the leaf bin
                pLeafBin->AddVisibleData( pElement->VBIndex,
                                          pElement->IndexStart,
                                          pElement->PrimitiveCount );
            } // Next Element

        } // Next RenderData Item

    } //  End if visible
}
```

The function first stores the passed visibility Boolean in the leaf's member variable which describes the leaf as either being visible or non visible. If the leaf has been flagged as visible then we need to add the triangles of every RenderData item stored there (one for each subset contained in the leaf) to their associated leaf bins. Notice how we loop through each RenderData structure stored in the leaf's array to add its data to its attached leaf bin. However, the triangles described by a single RenderData structure may be split over multiple vertex buffers (and multiple index buffers in the leaf bin). Therefore, for each RenderData structure we then loop through each RenderElement stored there. There will usually only be a single RenderElement structure stored in a RenderData structure. The exception to the rule occurs when, during the adding of this subset's polygon data to the vertex buffer, a new buffer had to be created such that the RenderData of a leaf for a given subset spanned vertex buffer boundaries. For each render element, we then call the leaf bin's AddVisibleData member. We looked at a single vertex buffer version of this function earlier, but let us now see how that function might look now that multiple lists of render batches will need to be maintained in the leaf bin for each vertex/index buffer pair.

Because the leaf bin no longer contains only a single index buffer and a single render batch list, we will need to introduce an additional structure in the leaf bin, which in this code is called CLeafBinData. This is a simple structure that pairs an index buffer with one of the tree's vertex buffers and stores its associated RenderBatch list during the visibility traversal. The CLeafBin class would now have just the following members (methods not shown):

```
class CLeafBin
{
    unsigned long  m_nAttribID;
    CLeafBinData **m_ppBinData;
    unsigned char  m_nVBCount;
};
```

The first member would describe the subset ID for which this leaf bin houses indices and compiles render batches for. The second is a pointer to an array of CLeafBinData objects. It is a CLeafBinData object that contains the pointers to an index buffer and vertex buffer pair and contains the RenderBatch

list for that buffer pair. As an example, if this leaf bin stored triangles for subset 6, and the vertices of polygons in the tree that use subset 6 are distributed across five vertex buffers, this leaf bin's m_nVBCount member would be set to 5 and would describe the number of elements in the CLeafBinData array. Each CLeafBinData structure then describes the index buffer and render batches used to render the triangles from those buffers.

The members of the CLeafBinData object would be as follows:

```
class CLeafBinData
{
   LPDIRECT3DVERTEXBUFFER9 m_pVertexBuffer;
   LPDIRECT3DINDEXBUFFER9  m_pIndexBuffer;
   unsigned long           m_nLastIndexStart;
   unsigned long           m_nLastPrimitiveCount;
   RenderBatch            *m_pRenderBatches;
   unsigned long           m_nBatchCount;
};
```

There is an important distinction to make between the vertex buffer pointer and the index buffer pointer in this structure. The vertex buffers are created and managed by the tree, not the leaf bins, so this is a pointer to a tree owned vertex buffer. There may be CLeafBinData items in multiple leaf bins that all point to this same vertex buffer as they all have triangles stored within it. The index buffers however are created and managed by the leaf bin and will not be shared by any other bins. That is, an index buffer contains the triangles for a specific leaf bin that are contained in a specific vertex buffer. Notice also that in our original discussion it was the leaf bin that managed the list of RenderBatch structures during the visibility pass. This is now stored in the CLeafBinData structure as we will need to compile render batch lists for each index/vertex buffer used by the bin. This is because we will have to execute the render batches for each index buffer only after we have bound it to the device along with its associated vertex buffer. Finally, the m_nLastPrimitiveCount and m_nLastIndexStart members have also been moved from the leaf bin into the CLeafBinData structure. These members were used during render batch building to remember the settings for the current batch being compiled between function calls. As we now have multiple index buffers in a single leaf bin, we will have to maintain multiple render batch lists and therefore, will need to store these 'm_nLast…' members on a per batch list basis.

We saw just a moment ago how the CBaseLeaf::SetVisible method calls the CLeafBin::AddVisibleData method to add the triangles of each RenderElement in each RenderData structure stored in the leaf. We had a look at a single vertex buffer version of this function earlier. Let us now see what it might look like with multiple vertex buffer support in the leaf bins.

```
void CLeafBin::AddVisibleData( unsigned char VBIndex,
                               unsigned long IndexStart,
                               unsigned long PrimitiveCount )
{
   CLeafBinData * pData = m_ppBinData[ VBIndex ];

   ULONG LastIndexStart       = pData->m_nLastIndexStart,
         LastPrimitiveCount   = pData->m_nLastPrimitiveCount;

   // Build up batch lists
```

```
   if ( LastPrimitiveCount == 0 )
   {
        // We don't have any data yet so just store initial values
        LastIndexStart     = IndexStart;
        LastPrimitiveCount = PrimitiveCount;

   } // End if no data
   else if ( IndexStart == (LastIndexStart + (LastPrimitiveCount * 3)) )
   {
        // The specified primitives were in consecutive order
        LastPrimitiveCount += PrimitiveCount;

   } // End if consecutive primitives
   else
   {
   // Store any previous data for rendering
   pData->m_pRenderBatches[pData->m_nBatchCount].IndexStart= LastIndexStart;
   pData->m_pRenderBatches[pData->m_nBatchCount].PrimitiveCount=LastPrimitiveCount;
   pData->m_nBatchCount++;

        // Start the new list
        LastIndexStart     = IndexStart;
        LastPrimitiveCount = PrimitiveCount;

   } // End if new batch

   // Store the updated values
   pData->m_nLastIndexStart     = LastIndexStart;
   pData->m_nLastPrimitiveCount = LastPrimitiveCount;
}
```

As you can see, the changes are subtle as the render batches are now compiled in the CLeafBinData structures instead of the CLeafBin object's themselves. The function's first parameter is the index of the tree's vertex buffer that the passed index start and primitive count value pertain to. We use the vertex buffer index to fetch the correct CLeafBinData structure from the leaf bin's array (there will be one for each vertex buffer being used by the tree) and from that point on, we are simply growing or starting new render batches in the CLeafBinData structures instead of in the leaf bin itself.

After the visibility process has been performed on the tree, each leaf bin may contain multiple render batches in each CLeafBinData structure. Each structure in this array describes the batches of the visible triangles that need to be rendered for a specific vertex/index buffer pair.

At this point, the application will choose to render the subsets of the tree. We saw earlier that the CBaseTree::DrawSubset call really just forwards the render request on the leaf bin that matched the passed subset ID. We did examine a version of a CLeafBin->Render function, but it lacked multiple vertex buffer support. Here is the new version of the function that instructs the leaf bin to render all render batches associated with all vertex/index buffers it contains.

```
void CLeafBin::Render( LPDIRECT3DDEVICE9 pDevice )
{
    ULONG i, j;
```

```
    // Loop through each vertex buffer
    for ( i = 0; i < m_nVBCount; ++i )
    {
        CLeafBinData * pData = m_ppBinData[ i ];
        if ( !pData ) continue;

        // Set the stream sources to the device
        pDevice->SetStreamSource( 0, pData->m_pVertexBuffer, 0, sizeof(CVertex) );
        pDevice->SetIndices( pData->m_pIndexBuffer );

        // Set the FVF
        pDevice->SetFVF( VERTEX_FVF );

        // Render the leaves
        for ( j = 0; j < pData->m_nBatchCount; ++j )
        {
            RenderBatch & Batch = pData->m_pRenderBatches[j];

            // Render any data
            pDevice->DrawIndexedPrimitive( D3DPT_TRIANGLELIST,
                                           0,
                                           0,
                                           pData->m_nVertexCount,
                                           Batch.IndexStart,
                                           Batch.PrimitiveCount );

        } // Next element

    } // Next Vertex Buffer
}
```

This function has the task of looping through each CLeafBinData structure in its array, binding its vertex and index buffers to the device and then looping through its render batches and rendering each one. Remember, the leaf bin's m_nVBCount member describes the number of vertex buffers in use by the tree and therefore describes the size of each leaf bin's CLeafBinData array. If the tree stored its geometry in five vertex buffers, but the leaf bin only contained triangles stored in the first vertex buffer, there would still be five CLeafBinData objects in its array, although the other four would have no render batches and no index buffers created or stored in them.

So, we loop through each CLeafBinData structure and for each one we process, we bind its vertex and index buffer pair to the device. We then loop through its render batches and call DrawIndexedPrimtive to render each batch. At the end of the outer loop we will have rendered all visible geometry in this leaf bin.

Although this sounds like a complicated system, it will become easier to grasp when we start to cover the actual code in the next section. Seeing how the leaf bins are constructed during the BuildRenderData call should clear up a lot of unanswered questions you may have at this point. The goal of this first section has been to provide you with some insight as to how the system will work and the various structures that will be used to implement it. This should make covering the rather tedious source code a lot easier.

It is now time to start discussing the real source code in the CBaseTree render system. As we have often done, we will cover the source code from a bottom up perspective. That is, we will look at the changes we will have to make at the CBaseLeaf level and filter those changes up until we finally cover the code changes to CBaseTree itself. To not do so would be pointless. For example, if we were to cover the CBaseTree::BuildRenderData method first, you will see that it spends most of its time calling member functions of CBaseLeaf and CLeafBin to accomplish its tasks. By looking at these support objects first, we can make sure that when we do finally get to examining the CBaseTree code, we understand all the function calls it is making and the support structures used.

# 15.3 CBaseLeaf - Adding Rendering Support

In the previous lesson we discussed much of the CBaseLeaf implementation and we listed and examined all code that related to building and querying the tree. In this section we will look at CBaseLeaf methods and member variables that were not discussed in the previous lesson as they pertained to the render system.

As the previous section indicated, part of the job of the CBaseTree::BuildRenderData method will be to store at each leaf a RenderData structure for each subset contained in that leaf. That is, if the polygon data in a given leaf uses three attributes (i.e., belongs to three different subsets), there will be an array of three RenderData items stored at that leaf.

We also saw that because we need to support multiple vertex buffers, each RenderData item cannot just store a single index start and primitive count since the associated leaf bin will also need to know in which of the tree's vertex buffers the triangles are contained. This also indicates which index buffer that the leaf will have stored these triangles in within the leaf bin. The RenderData structure represents a single subset within a leaf. It contains a pointer to the associated leaf bin that collects triangles for that subset and also contains an array of render elements (Element structures).

## 15.3.1 The RenderData Structure

The RenderData structure is shown below along with the Element structure which is contained within its namespace. Its definition is contained in CBaseTree.h. The RenderData structure is actually defined inside the CBaseLeaf namespace, but for the sake of clarity in this discussion, we show it outside that namespace.

```
struct RenderData
{
        struct Element
        {
            unsigned long   IndexStart;         // Index of this leaves first tri
            unsigned long   PrimitiveCount;     // Number of Tris
            unsigned char   VBIndex;            // vertex Buffer Index
        };
```

```
        unsigned long AttributeID;    // The attribute ID of this render data item
        CLeafBin    * pLeafBin;       // Pointer to the actual leaf bin
        Element     * pElements;      // The actual render data element array
        unsigned long ElementCount;   // The number of elements stored here.
};
```

Let us discuss the four members of this structure.

**unsigned long AttributeID**
This member stores the subset ID of the triangles in the leaf that this RenderData structure represents. This will match the attribute ID of the leaf bin pointed at by its pLeafBin member.

**CLeafBin * pLeafBin**
This member stores a pointer to the leaf that is to collect the triangles in this RenderData structure during a visibility pass if the leaf in which the RenderData structure is contained is deemed visible. We store the pointer to the leaf bin in the RenderData structure so that we can easily call its AddVisibleData method to send it the render elements of this RenderData structure.

**Element * pElements**
This is a pointer to an array of Element structures. Each element describes a run of triangles that needs to be added to the leaf bin during a visibility pass, along with a vertex buffer index. The vertex buffer index informs the leaf bin's AddVisibleData methods which render batch list the triangles in that element should be added to. Most of the time, a single RenderData structure will contain only one Element in this array which describes (to the associated leaf bin) which vertex buffer/index buffer pair these triangles are associated with and therefore, which render batch list the triangles should be added to in that leaf bin.

The only time when there will exist more than one Element in a leaf's RenderData structure will be if, during the BuildRenderData method, a vertex buffer switch had to be made partway through adding a leaf's subset to a leaf bin. For example, we know that if we have 50 polygons that use subset 5, during the build phase the vertices of these polygons will need to be added to the tree's vertex buffer and the indices of these 50 triangles should be added to the leaf bin's index buffer. Under normal circumstances were no complications to occur, we would then store (inside the RenderData structure for that leaf's subset) a primitive count of 50 and an index start describing where this leaf's run of triangles start in that leaf bin's index buffer. However, what if after adding only 25 triangles during the build process, the vertex buffer becomes full and the remaining 25 of this leaf's subset 5 polygons have to be added to a new vertex buffer? The leaf bin will be informed when the second 25 triangles are added that these exist in the second vertex buffer, not the first. The leaf bin would then create a second index buffer that will be used to index the triangles in the second vertex buffer. If we go back to the leaf and subset in question, we now have a RenderData structure for subset 5 in a leaf that is spread over two vertex and index buffers. Therefore, in such a situation two Element structures will be stored in its RenderData structure's pElements array. The first element will describe the array of subset 5 triangles in the first vertex/index buffer and the second element will describe the second run of subset 5 triangles in the second vertex/index buffer. As we discussed, during the visibility pass, each element will be passed to the leaf bin which will maintain render batch lists for each vertex/index buffer combination it uses.

**unsigned long ElementCount**
This member describes the number of elements stored in the previously described array. As discussed, this will usually be 1 unless the polygons in this leaf that belong to the subset represented by this RenderData structure span multiple vertex buffers.

## 15.3.2 CBaseLeaf – The Source Code

CBaseLeaf is no stranger to us as it was covered quite heavily in the previous chapter. In this chapter however we will cover its methods and members pertaining to the rendering system. Although we show the entire class below contained CBaseTree.h (minus dynamic object support which will be added later in the lesson), you will see that there are only a handful of new methods and members that we did not cover in the previous lesson. These are highlighted in bold.

```
class CBaseLeaf : public ILeaf
{
public:

    // Constructors & Destructors for This Class.
    virtual ~CBaseLeaf( );
             CBaseLeaf( CBaseTree * pTree );

    // Public Virtual Functions for This Class (from base).

    virtual bool            IsVisible           ( ) const;

    virtual unsigned long   GetPolygonCount     ( ) const;
    virtual CPolygon *      GetPolygon          ( unsigned long nIndex );
    virtual unsigned long   GetDetailAreaCount  ( ) const;
    virtual TreeDetailArea* GetDetailArea       ( unsigned long nIndex );
    virtual void            GetBoundingBox      ( D3DXVECTOR3 & Min,
                                                  D3DXVECTOR3 & Max ) const;

    // Public Functions for This Class.
    void                    SetVisible          ( bool bVisible );

    void                    SetBoundingBox      ( const D3DXVECTOR3 & Min,
                                                  const D3DXVECTOR3 & Max );
    bool                    AddPolygon          ( CPolygon * pPolygon );
    bool                    AddDetailArea       ( TreeDetailArea * pDetailArea );

    RenderData            * AddRenderData       ( unsigned long nAttribID );
    RenderData::Element   * AddRenderDataElement( unsigned long nAttribID );
    RenderData            * GetRenderData       ( unsigned long nAttribID );


protected:


    // Protected Structures, Enumerators and typedefs for This Class.
```

```
    typedef std::vector<CPolygon*>          PolygonVector;
    typedef std::vector<TreeDetailArea*>    DetailAreaVector;


    // Protected Variables for This Class

    PolygonVector       m_Polygons;         // Array of leaf's polygon pointers
    DetailAreaVector    m_DetailAreas;      // Array of detail area pointers.
    bool                m_bVisible;         // Is this leaf visible or not?
    D3DXVECTOR3         m_vecBoundsMin;     // Minimum bounding box extents
    D3DXVECTOR3         m_vecBoundsMax;     // Maximum bounding box extents.

    RenderData         *m_pRenderData;      // Renderable data information
    unsigned long       m_nRenderDataCount; // Number of render data items .

    CBaseTree          *m_pTree;            // The tree to which this leaf belongs.
};
```

There are only three new members that have been added to the leaf which are used to store the RenderData structures.

**RenderData *m_pRenderData**
This array will contain a RenderData structure for each subset of polygons that exist in the leaf. The RenderData structure was discussed above and contains the runs of triangles (called elements) stored in the leaf bin's index buffers. If a leaf contains polygons that belong to five different subsets, the leaf will store five RenderData structures in this array. We can think of each RenderData structure as being an input of visible triangles into a specific leaf bin when the leaf is found to be visible.

**unsigned long m_nRenderDataCount**
This member describes the number the RenderData structures in the above array. This implicitly tells us the number of subsets contained in this leaf and the number of leaf bins we will have to provide render elements for if this leaf is found to be visible.

**bool m_bVisible**
This Boolean is set to either true or false by the visibility determination process. If the ProcessVisibility method of the spatial tree determines that this leaf is inside the view frustum, it will be set to true; otherwise it will be set to false.

Notice that CBaseLeaf also has several new methods which relate to the visibility system. These methods will never be called by the application, as they are used by the tree when building and preparing the render data and processing tree visibility. We will discuss each of these methods next.

## IsVisible – CBaseLeaf

This method is an exception to the rule with respect to the other leaf visibility methods in that it will often be used by the application to query the visibility status of a leaf. For example, the application may send an AABB down the tree to collect all leaves that fall within that bounding volume. The CollectLeavesAABB method would be used to fill a leaf list with these CBaseLeaf pointers. The application could then iterate through the returned leaf list and call the IsVisible method to determine if that leaf is currently visible and needs to be processed.

The method itself is a simple one line function that returns the value of its m_bVisible member variable. It will contain the visibility status of the leaf as calculated in the last visibility pass through the tree (i.e., the most recent call to CBaseTree::ProcessVisibility).

```
bool CBaseLeaf::IsVisible( ) const
{
    return m_bVisible;
}
```

## AddRenderData – CBaseLeaf

The CBaseLeaf::AddRenderData method should never be called by the application. It is only called by the spatial tree during the BuildRenderData call when the data is being prepared in its renderable form. It is at this stage (just after the tree has been built) that each leaf is processed and the RenderData structures are generated and stored in each leaf (one for each subset contained in that leaf).

Since the leaf stores its RenderData structures in an array, this array will have to be grown every time we wish to add a new RenderData structure to it during the building of the render data. Remember, at the start of the BuildRenderData call, no leaves will contain any RenderData structures. These will need to be allocated and added to a leaf's RenderData array as and when we examine the polygons of that leaf and find a polygons in that leaf with an attribute ID which we have not yet generated a RenderData structure for. The CBaseTree::BuildRenderData method will call this method when it would like to make room in the leaf's RenderData array for a new structure. The function is passed the attribute ID of the subset that this RenderData structure represents in the current leaf. The function will resize the leaf's RenderData structure array and store the attribute ID in the new RenderData structure. A pointer to this new element in the array will then be returned to the function so that it can be used to populate the new RenderData structure with render elements.

This code is like most we have seen that resizes arrays. It first allocates a new empty array of RenderData structures that is large enough to store the number of elements currently in the RenderData array plus the one new element we wish to add. We then copy over all the data from the previous array into the new array before releasing the old array and assigning the leaf's pointer to point at this new array instead. We also store the passed attribute ID in the newly added RenderData structure so that we know which subset set this RenderData structure will store polygons for.

```
CBaseLeaf::RenderData * CBaseLeaf::AddRenderData( unsigned long nAttribID )
{
    RenderData * pBuffer, * pData;

    // First allocate space for all the element items
    pBuffer = new RenderData[ m_nRenderDataCount + 1 ];
    if ( !pBuffer ) return NULL;

    // Any old data?
    if ( m_nRenderDataCount > 0 )
    {
        // Copy over the old data, and release the old array
        memcpy( pBuffer, m_pRenderData, m_nRenderDataCount * sizeof(RenderData) );
        delete []m_pRenderData;

    } // End if any old data

    // Get the element we just created
    pData = &pBuffer[ m_nRenderDataCount ];

    // Clear the new entry
    ZeroMemory( pData, sizeof(RenderData) );

    // Store the attribute ID, it's used to look up the render data later
    pData->AttributeID = nAttribID;

    // Store the new buffer pointer
    m_pRenderData = pBuffer;
    m_nRenderDataCount++;

    // Return the item we created
    return pData;
}
```

Notice that when this new RenderData structure is added to the array it is initially empty. It contains no Elements in its pElements array, which means initially it will be linked to no leaf bin and will not contain any triangles. The calling function will use the returned pointer to set its leaf bin pointer and add Elements to it.

## GetRenderData - CBaseLeaf

Although the application should never need to call a function such as this, it will be used by the BuildRenderData method when it wishes to retrieve a pointer to one of the leaf's RenderData structures. A classic example of this is when the BuildRenderData method is processing the polygons in a given leaf. If it finds a polygon that uses subset 5 for example, it will call the GetRenderData method passing in an attribute ID of 5. If a valid pointer is returned, then it means we have already processed triangles in this leaf for the same subset and as such, its RenderData structure has already been allocated. When this is the case, we can just increment the primitive count of the RenderData structure and continue on to the next polygon in the leaf. If a NULL pointer is returned, it tells the BuildRenderData method that we are currently processing a polygon in a leaf that belongs to a subset we have not yet found in that leaf. This

means that there will not currently be a RenderData structure allocated for it. When this happens, the BuildRenderData method can call the CBaseLeaf::AddRenderData method to add a new RenderData structure to the leaf's array. The current index start and primitive count of this polygon (and any future ones we find in this leaf which share the same subset as this polygon) can then be appropriately stored.

The key for the search is the subset ID (attribute ID) of the RenderData structure you are looking for within this leaf. The function loops through the leaf's array of RenderData structures searching for one with a matching attribute ID. If one is found, its pointer is returned. Otherwise, the loop plays out to its conclusion and we return NULL. This indicates that no RenderData item currently exists in the leaf which matches the passed attribute ID.

```
CBaseLeaf::RenderData * CBaseLeaf::GetRenderData( unsigned long nAttribID )
{
    ULONG i;

    // Search for the correct render data item
    for ( i = 0; i < m_nRenderDataCount; ++i )
    {
        if ( m_pRenderData[i].AttributeID == nAttribID ) return &m_pRenderData[i];
    } // Next RenderData Item

    // We didn't find anything
    return NULL;
}
```

You will see many of these methods being used when we cover the CBaseTree::BuildRenderData method later in the lesson.

## AddRenderElement - CBaseLeaf

This method is another method that will never be called by the application but will be used by the CBaseTree::BuildRenderData method while constructing the RenderData structures at each leaf.

As discussed previously, each RenderData structure stored in a leaf will represent that leaf's polygon contribution for a single subset. Each RenderData structure is used to describe a block of triangles in the associated leaf bin's index buffer(s). Usually, each RenderData structure will store a single Element where each element represents the block of triangles in a given leaf bin's index buffer. However, if when adding the vertices of a given leaf's subset, the vertex buffer is found to be full, a new vertex buffer will be generated and that leaf's subset will be split over multiple vertex buffers. When this is the case, we must store multiple Elements in the RenderData structure so that we can inform the associated leaf bin during the visibility pass about which vertex/index buffers store all the triangles for this leaf. For example, if this leaf has polygons that belong to subset 10, but they are spread over three different vertex buffers, the leaf's RenderData structure for subset 10 will contain three Elements. Each Element describes the vertex buffer and index buffer this run of triangles is stored in and the location and number of indices in those buffers.

Because we never know before building how many Elements we will need to store in each leaf, we will need a way for the BuildRenderData method to add new Elements to a leaf's RenderData item structure as and when we encounter a situation where one has to be added. The AddRenderElement method simply resizes the Element array of the leaf's RenderData structure that is associated with the passed subset ID making room at the end for a new Element structure. A pointer to this new Element is returned to the calling function so that it can populate the new Element structure with index start and primitive count information for a given vertex buffer. We will see how this function is used later.

The method is passed a subset ID as its only parameter. The Leaf's GetRenderData method is then called to retrieve a pointer to the RenderData structure within the leaf that represents triangles assigned with the passed attribute.

```
CBaseLeaf::RenderData::Element * CBaseLeaf::AddRenderDataElement
                                                ( unsigned long nAttribID )
{
    RenderData::Element * pBuffer, * pElement;
    RenderData * pData = GetRenderData( nAttribID );
    if ( !pData ) return NULL;
```

If the GetRenderData method returned NULL, it means the caller is trying to add an Element to the array of a RenderData structure which has not yet been created. In this case, the function returns NULL. This informs the caller that there currently exists no RenderData structure in the leaf associated with the passed subset ID.

At this point we allocate a new Element array (notice it is within the RenderData namespace) large enough to store any Elements that may already be in the array plus the new one we wish to add. If there are any elements already in the previous array of the RenderData structure, we copy them over into the new array we have just allocated. We then delete the old array previously pointed at by the RenderData::pElements pointer.

```
    // First allocate space for all the element items
    pBuffer = new RenderData::Element[ pData->ElementCount + 1 ];
    if ( !pBuffer ) return NULL;

    // Any old data?
    if ( pData->ElementCount > 0 )
    {
        // Copy over the old data, and release the old array
        memcpy( pBuffer,
                pData->pElements,
                pData->ElementCount * sizeof(RenderData::Element) );

        delete []pData->pElements;

    } // End if any old data
```

At this point we store a pointer to the new Element structure we added at the end of the new array so we can return the structure to the caller.

```
    // Get a pointer to the new element
    pElement = &pBuffer[ pData->ElementCount ];
```

The new Element structure at the end of the array is un-initialized, so we clear its memory.

```
    // Clear the new entry
    ZeroMemory( pElement, sizeof(RenderData::Element) );
```

We then assign the RenderData structure's pElements pointer to point at our new array (as the old one has been deleted) and increase the RenderData::ElementCount member variable to correctly reflect the number of elements in the array. Finally, we return the pointer to the new element at the end of this array which the caller will then use to populate with meaningful data.

```
    // Store the new buffer pointer
    pData->pElements = pBuffer;
    pData->ElementCount++;

    // Return the element we created
    return pElement;
}
```

## SetVisible - CBaseLeaf

We have seen a few simplified versions of this function during our initial discussions of the render system, but now we will see the real thing. Before we look at the code, you should be aware of a new member that has been added to CBaseTree.

We will see later that CBaseTree now has an additional LeafList member variable which is declared as shown below:

Excerpt from CBaseTree.h ( CBaseTree class )

```
LeafList                    m_VisibleLeaves;
```

Remembering that LeafList is a type definition for an STL list that contains ILeaf pointers, the name of this member should imply what it is used to store.

When the visibility pass is performed on the tree, a traversal method in the derived class will be used to walk through the tree finding visible leaves. At the very start of this visibility traversal (which is performed with each frame update), the m_VisibleLeaves list will be emptied. Once a visible leaf has been found during traversal, the leaf's SetVisible method will be called. This function will set the leaf's visibility Boolean to true. It will also loop through every RenderData structure stored in the leaf and add each one's Elements to their associated leaf bins as we discussed earlier. All of this code has been covered earlier and is shown below.

```
void CBaseLeaf::SetVisible( bool bVisible )
{
```

```
    ULONG                  i, j;
    RenderData::Element * pElement;
    CLeafBin            * pLeafBin;
    RenderData          * pData;

    // Flag this as visible
    m_bVisible = bVisible;

    // If we're being marked as visible, inform the renderer
    if ( m_bVisible && m_nRenderDataCount > 0 )
    {
        // Loop through each renderable set in this leaf.
        for ( i = 0; i < m_nRenderDataCount; ++i )
        {
            pData    = &m_pRenderData[i];
            pLeafBin = pData->pLeafBin;

            // Loop through each element to render
            for ( j = 0; j < pData->ElementCount; ++j )
            {
                pElement = &pData->pElements[j];
                if ( pElement->PrimitiveCount == 0 ) continue;

                // Add this to the leaf bin
                pLeafBin->AddVisibleData( pElement->VBIndex,
                                          pElement->IndexStart,
                                          pElement->PrimitiveCount );
            } // Next Element

        } // Next RenderData Item

    } //  End if visible
```

However, the final part of this function is new. If the visibility status of the tree is set to true, the leaf adds its own pointer to the tree's m_VisibleLeaves list using a new method of CBaseTree called AddVisibleLeaf. This is a simple method which just wraps the adding of the passed leaf pointer to the tree's visible leaf list. The remainder of this function is shown below.

```
    // Update tree object's if we're visible
    if ( m_bVisible )
    {
        // // Add this leaf to the tree's visible leaf list
        m_pTree->AddVisibleLeaf( this );

    } // End if we are visible
}
```

After the visibility process has been performed for a given frame update, and the SetVisible method has been called for all visible leaves, the spatial tree's m_VisibleLeaves list will contain pointers for all leaves which are currently visible. The application can fetch this list by calling the spatial tree's GetVisibleLeaves method.

It is very useful for the application to be able to ask the tree for a list of currently visible leaves. For example, the application can loop through these leaves and only bother rendering the dynamic objects contained in those leaves. Any dynamic object which is stored in non-visible leaves will simply be ignored by the rendering loop. After we have discussed the rendering system for the static geometry of the spatial tree, we will discuss how the application can assign its own dynamic objects to leaves within the spatial tree and update their positions as they moves around the world.

We have now discussed all the changes to CBaseLeaf that have been added to facilitate the needs of the rendering system. Most of the methods that we have added are utility methods that will be used by the tree's BuildRenderData method to add and retrieve RenderData structures and Element structures to the leaves as the data is being added to leaf bins.

## 15.3.3 Leaf Bins

When we cover the rendering enhancements to CBaseTree in a moment, we shall see that one of its new members is an array of CLeafBin structures. A leaf bin is an object that stores the indices of all triangles in the tree that use a specific subset. If multiple vertex buffers are being used by the tree to contain the entire scene, and the subset for a specific leaf bin has geometry spread over those multiple buffers, the leaf bin will contain an index buffer for each vertex buffer being used by that subset. The number of index buffers being used is not necessarily equal to the number of vertex buffers being used by the tree. If the tree's static geometry is contained in five vertex buffers, but subset 6 (for example) had triangles contained in only two of those buffers, the leaf bin for subset 6 will contain only two index buffers. Although the leaf bin need not necessarily use a separate index buffer for each vertex buffer containing triangles of its associated subset, it makes the system cleaner and easier to understand. We know for example that if a leaf bin has triangles in vertex buffers 1 and 3, when the leaf bin is rendered, we will need to set vertex buffer 1 first and its associated index buffer and execute the render batches stored for that buffer combination. Then we will then have to set the second index/vertex buffer pair and execute the render batches relative to that buffer set.

In our earlier discussions, we also discovered that a leaf bin is not just a place where all the static index buffers used by a given subset of polygons reside; it also has methods and structures used during the visibility pass to construct a list of render batches for each buffer combination. These render batches describe which triangles in that leaf bin are currently considered visible by the system and should be rendered when the application wishes to render that subset of the tree.

Each render batch structure represents a block of continuous visible triangles in one of the leaf bin's index buffers, and as such describes a block of triangles in that leaf bin that can be rendered with a single draw call. Because the currently visible triangles stored in a leaf bin may not all be arranged in consecutive order in the leaf bin, many render batches will typically be created representing all the visible sections of the index buffer that need to be rendered. Furthermore, because a leaf bin may have to manage multiple index buffers due to the fact that its triangle data is spread over multiple vertex buffers, a render batch list will need to be constructed for each buffer combination. The render batch lists are destroyed and rebuilt for each leaf bin every time a visibility pass of the tree is performed. We can just

think of a render batch list as simply being a list of drawing instructions for a given buffer pair inside the leaf bin.

CBaseTree will now manage a list (actually an STL map, but more on this in a moment) of leaf bins. The number of leaf bins allocated for the tree will be equal to the number of subsets used by the static polygon data stored in the tree. That is, if the polygon data in the tree belongs to a combination of 15 subsets, the BuildRenderData method will create 15 leaf bins, one for each subset. The BuildRenderData method will then proceed to fill those leaf bins with the indices of the triangles that belong in each leaf bin.

We have already seen the CLeafBin::AddVisibleData method being used by the CBaseLeaf::SetVisible method and we will see many of its other methods being used later during the building of the render data. For now though, we will examine the CLeafBin object, look at the structures and support objects it uses, the methods it exposes and the code to those methods.

## 15.3.4 The RenderBatch Structure

During the visibility pass, the leaf bin will build a list of RenderBatch structures for each buffer combination used by the leaf bin's triangles. Each RenderBatch structure simply describes a block of triangles in one of the leaf bin's index buffers that can be rendered with a single draw call.

```
struct RenderBatch
{
    unsigned long IndexStart;
    unsigned long PrimitiveCount;
};
```

There will be a RenderBatch list constructed during the visibility pass for each index buffer in use by the leaf bin.

**unsigned long IndexStart**
This value will contain the position in the associated index buffer of the first index in this run of visible triangles.

**unsigned long PrimtiveCount**
This describes how many triangles starting at IndexStart are contained in this visible continuous section of the index buffer.

## 15.3.5 CLeafBinData – The Source Code

A CLeafBin object contains an array of CLeafBinData structures. Each CLeafBinData structure houses a vertex/index buffer combination, information about those buffers, and the RenderBatch list for that vertex and index buffer pair. Rendering a leaf bin involves looping through its CLeafBinData array and setting the vertex and index buffers stored in the current CLeafBinData element being processed before looping through the CLeafBinData's render batch list calling DrawIndexedPrimitive for each one.

If the spatial tree is using five vertex buffers, every leaf bin will contain an array of five CLeafBinData objects as this is the maximum number of vertex/index buffers that could ever be needed by a leaf bin. However, not all the elements in the CLeafBinData array are necessarily used by the leaf bin. If the leaf bin represents geometry that is spread over only two of the five vertex buffers in use by the tree, that leaf bin will only use the first two elements in its CLeafBinData array. Each element in that array would contain the index buffer and vertex buffer to set on the device in order to render the batches for that buffer combination.

The CLeafBinData class is really just a structure with a constructor that initializes its members to zero. Although it has a fair number of members, they really just describe a vertex buffer and an index buffer and information about those buffers (e.g., the amount of data that is contained in them).

This object is declared in CBaseTree.h and is shown below followed by a description of its members.

```
class CLeafBinData
{
public:

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

    // Public Variables for This Class.
    LPDIRECT3DVERTEXBUFFER9 m_pVertexBuffer;
    LPDIRECT3DINDEXBUFFER9  m_pIndexBuffer;

    unsigned long           m_nFaceCount;
    unsigned long           m_nVertexCount;
    unsigned char           m_nVBIndex;
    bool                    m_b32BitIndices;
    unsigned long           m_nLastIndexStart;
    unsigned long           m_nLastPrimitiveCount;
    RenderBatch            *m_pRenderBatches;
    unsigned long           m_nBatchCount;
};
```

**LPDIRECT3DVERTEXBUFFER9 m_pVertexBuffer**
This member stores a pointer to the tree's vertex buffer for which this object is applicable. The tree is responsible for allocating and releasing this vertex buffer; this is just a pointer to it so that we can easily

bind it to the device before rendering the RenderBatch instructions stored here. The vertex buffer is not specific to any one leaf bin and may be shared by many leaf bins. All leaf bins that have geometry in one of the tree vertex buffers will all have a CLeafBinData structure that has a pointer to that buffer.

**LPDIRECT3DINDEXBUFFER9 m_pIndexBuffer**
This is a pointer to an index buffer that contains all the triangles in the above vertex buffer that use the subset associated with the parent leaf bin. This index buffer is not shared by any other leaf bins or even any other CLeafBinData structures within the same leaf bin. This is a static index buffer that is built at leaf bin creation time and it contains all geometry (not just visible geometry) that uses the applicable subset of the leaf bin.

**unsigned long m_nFaceCount**
Describes how many triangles are in the above index buffer.

**unsigned long m_nVertexCount**
Describes how many vertices are contained in the vertex buffer pointed to by the first member of this structure.

**unsigned char m_nVBIndex**
This member describes the vertex buffer pointed to by the first member of this structure as a numerical index in the spatial tree's vertex buffer array. For example, if the spatial tree has stored its geometry in five vertex buffers, but this CLeafBinData structure was representing triangle data stored in the second vertex buffer, this value would be set to 1 (the index of the second vertex buffer in the spatial tree's vertex buffer array).

**bool m_b32BitIndices**
This member is set to true or false when building the leaf bins to determine whether the index buffer stored here uses 32-bit indices (rather than 16-bit indices). This is certainly useful information to have if we ever wish to lock the index buffer and access the data.

**unsigned long m_nLastIndexStart**
We saw this member being used when we looked at a simplified version of the leaf bin's AddVisibleData method. It records the first index in a batch of adjacent triangles currently being collected. This value is first set when the AddVisibleData method is called and must start collecting for a new render batch. Every time the function is called after that, if the passed run of adjacent triangles follows on from the ones added to the batch in the previous function call, this value remains unaltered. When we do finally get passed some render data which does not form a continuous block with all the triangles we have collected so far (starting at m_nLastIndexStart), we have collected all we can for that batch and the triangles are committed to a new RenderBatch structure. m_nLastIndexStart is then set to the location of the new run of triangles we are starting to collect.

In short, this value is used as temporary storage during the visibility pass by the leaf bin to record the start of the current batch of adjacent indices collected so far.

**unsigned long m_nLastPrimitiveCount**
This member is also used as temporary storage during the visibility pass by the CLeafBin::AddVisibleData method. It is used to record, with each function call, the number of adjacent triangles we have collected so far for the current batch being compiled. We discussed this process in some detail earlier when we examined the construction of the RenderBatch lists.

**RenderBatch *m_pRenderBatches**
This member is an array of RenderBatch structures that will be flushed and filled every time a visibility pass is executed on the tree. This array is allocated (when the CLeafBinData item is first created) to be large enough to hold the maximum number of render batches that we could ever need to create. Although this admittedly wastes some memory, it does allow us to forego any array resizing overhead when building and storing the render batches. As this should be a critically fast process, we want to do everything we can to eliminate such overhead.

This array describes a list of draw instructions for the vertex and index buffers referenced by this structure. Each element in this array represents one DrawIndexedPrimitive call using the above vertex and index buffers.

**unsigned long m_nBatchCount**
The member stores the number of *valid* RenderBatch structures contained in the above array. Remember, the size of the above array never changes, as discussed above. It is allocated once when the leaf bin first creates this CLeafBinData item and is made large enough to contain the maximum number of render batches that could possibly be created (more on this in a moment). Therefore, this member really describes the number of elements in the above array that are currently considered valid with respect to the last visibility pass. Whenever a new visibility pass is performed, this member is initially set to zero and then incremented every time we add new RenderBatch information. Therefore, this value describes not only the number of batches in the above array that will need to be executed to draw all the visible data in the associated index buffer, but it will also describe the number of DrawIndexedPrimitive calls that will need to be performed to render that data.

## 15.3.6 CLeafBin – The Source Code

The object that binds this whole system together is CLeafBin. This class has very few member variables as it is really just a wrapper around an array of CLeafBinData structures for a given subset ID. It also has methods that allow you to retrieve the subset/attribute ID of the leaf bin, add and retrieve CLeafBinData structures from its array, and a method that renders the bin. The leaf bin also has the now familiar AddVisibleData method that is called by the CBaseLeaf::SetVisible method to add a leaf's polygon contribution to a render batch. The leaf bins themselves will be created by CBaseTree during the BuildRenderData method.

Below we show the class declaration for CLeafBin which is contained in CBaseTree.h. This is followed by an explanation of its member variables.

```
class CLeafBin
{
public:

    // Constructors & Destructors for This Class.
    virtual ~CLeafBin();
    CLeafBin( unsigned long nAttribID, unsigned long nLeafCount );

    // Public Functions for This Class.
    unsigned long   GetAttributeID      ( ) const;
    CLeafBinData  * GetBinData          ( unsigned long Index ) const;
    bool            AddLeafBinData      ( CLeafBinData * pData );
    void            AddVisibleData      ( unsigned char VBIndex,
                                          unsigned long IndexStart,
                                          unsigned long PrimitiveCount );
    void            Render              ( LPDIRECT3DDEVICE9 pDevice );

private:

    // Private Variables for This Class.
    unsigned long  m_nAttribID;     // The attribute ID of the faces stored here
    unsigned long  m_nLeafCount;    // Total number of leaves in the tree
    CLeafBinData **m_ppBinData;     // An array containing all the information
                                    // we need to render all the data in this bin
                                    // for each vertex bubber (one per primary VB).
    unsigned char  m_nVBCount;      // Number of vertex buffers in use by system
};
```

**unsigned long m_nAttribID**
This value of this member will be passed into the constructor when the leaf bin is first created by the tree. It contains the subset ID assigned to this leaf bin. That is, all the triangles in the tree that have a matching attribute ID will have their indices stored in this leaf bin.

**unsigned long m_nLeafCount**
The value of this member is also passed into the constructor when the leaf bin is first created by the tree. It contains the number of leaves in the tree. You will see why we need this value in a moment when we look at the remaining code.

**CLeafBinData **m_ppBinData**
This is a pointer to an array of CLeafBinData pointers. The size of this array will be equal to the number of vertex buffers being used by the tree. That is, it will hold enough pointers in each leaf bin for the possibility that the leaf bin might have triangles contained in every one of the vertex buffers and would therefore need to represent these triangles with a CLeafBinData structure for each.

Of course, this does not mean that we always allocate a CLeafBinData object for every vertex buffer for each leaf bin, since a given leaf bin might only use two of the vertex buffers and would therefore only need to have two CLeafBinData structures allocated for it. If a leaf bin does not have geometry in one or more of the tree's vertex buffers, the elements in this array that correspond to that vertex buffer will simply be set to NULL. For example, if the tree uses five vertex buffers to store its geometry, the m_ppBinData array in each leaf bin would be allocated large enough to store five pointers. However, if

a leaf bin only has geometry in the first and third vertex buffer, it would have only two CLeafBinData objects allocated for it and have their pointers stored in elements [0] and [2] in this array. The rest of the elements would be set to NULL for the other vertex buffers.

**unsigned char m_nVBCount**
This value describes the number of vertex buffers in use by the parent tree. This is useful because it also describes the size of the above array. We need this value so that when we render the leaf bin, we know how many element we have to loop through in the above array testing for valid pointers.

Let us now take a look at the methods for this class, which are for the most part, all rather small and straightforward. We will start with the constructor.

## Constructor - CLeafBin

The constructor is called by the tree during the generation of its renderable data. The application should never want to allocate a leaf bin. The constructor takes two parameters; the attribute ID of the subset this leaf bin will contain indices for, and the total leaf count of the tree for which the leaf bin is being created. These are stored in the corresponding member variables.

```
CLeafBin::CLeafBin( unsigned long nAttribID, unsigned long LeafCount )
{
    // Store passed values
    m_nAttribID  = nAttribID;
    m_nLeafCount = LeafCount;
    m_ppBinData  = NULL;
    m_nVBCount   = 0;
}
```

Notice that when the leaf bin is first created, the CLeafBinData pointer array is set to NULL and the vertex buffer count member is set to zero. The CLeafBinData structures will be added to the leaf bin (via the CLeafBin::AddLeafBinData method) once the leaf bin has been constructed.

## GetAttributeID – CLeafBin

This is a simple accessor function that allows the caller to query the attribute ID of the leaf bin to determine which subset of polygons it contains.

```
unsigned long CLeafBin::GetAttributeID( ) const
{
    return m_nAttribID;
}
```

## AddLeafBinData - CLeafBin

This method is called by the BuildRenderData method when it wishes to add a CLeafBinData object to the leaf bin's internal array. You will see in a moment when we discuss the CBaseTree::BuildRenderData method that it essentially loops through each leaf and then through each polygon in those leaves to build an array of vertices and an array of indices into those vertices for each leaf bin. Once the number of vertices we have collected exceeds the number supported in the single vertex buffer by the hardware, we will create the vertex buffer and fill it with the vertices we have collected so far. We will then create index buffers for each leaf bin that index into that vertex buffer. It is at this point that we will inform each leaf bin that we would like to add a new CLeafBinData item to its array which we will populate with the new vertex buffer and index buffer pointers.

The method itself is fairly routine. We pass in a pointer to a pre-allocated CLeafBinData object and the function will resize the leaf bin's array to make room for this pointer at the end. The first part of the function is like every other array resizing method we have discussed, so this should need no explanation.

```
bool CLeafBin::AddLeafBinData( CLeafBinData * pData )
{
    CLeafBinData ** ppBuffer;

    // First allocate space for all the data items
    ppBuffer = new CLeafBinData*[ m_nVBCount + 1 ];
    if ( !ppBuffer ) return false;

    // Any old data?
    if ( m_nVBCount > 0 )
    {
        // Copy over the old data, and release the old array
        memcpy( ppBuffer, m_ppBinData, m_nVBCount * sizeof(CLeafBinData*) );
        delete []m_ppBinData;

    } // End if any old data

    // Set the new entry
    ppBuffer[ m_nVBCount ] = pData;

    // Store the new buffer pointer
    m_ppBinData = ppBuffer;
    m_nVBCount++;
```

At this point we have resized the array and added the passed CLeafBinData pointer to the end of the array. However, what this function will also be responsible for is allocating the passed CLeafBinData structure's RenderBatch array. You will recall from our coverage of the CLeafBinData structure that the RenderBatch array will be allocated just once when the CLeafBinData structure is first added to the leaf bin. It will be made large enough to store the maximum possible number of render batches that could ever be generated for a leaf bin. This way, we will not have to suffer array resizes and flushes as the number of visible leaves fluctuates with each visibility update.

Because the indices of our triangles will be added to the leaf bin index buffers in tree traversal order to help minimize the number of render batches needed, the worst case is that our queue will contain batches where one leaf is visible and the next is invisible, repeating over the entire set. In such a scenario we would need to create a RenderBatch structure for every visible leaf, which in this scenario would half of them. This means that the highest number of batches we will ever need to store in a leaf bin is simply the total number of leaves divided by two. However, as we are performing a division using integer math, we will add one to the total leaf count prior to performing the division. This is to ensure that the result is rounding up and not down. We want 5/3=1.666 to be rounded up to 2 not down to 1. This will make sure the render batch array in each CLeafBinData structure is large enough to cope with the worst case visibility scenario.

Here is the remaining code that allocates the RenderBatch array for the passed CLeafBinData structure before returning.

```
    // Note: We add '1' to ensure that we don't have any rounding issues.
    pData->m_pRenderBatches = new CLeafBinData::RenderBatch[(m_nLeafCount+1) / 2 ];

    if ( !pData->m_pRenderBatches ) return false;

    // Success!
    return true;
}
```

## GetBinData - CLeafBin

This method is a simple function that allows the caller to retrieve one of the leaf bin's CLeafBinData pointers. The caller simply passes the index of the element in the array it would like to retrieve the CLeafBinData item for. This may be NULL if the leaf bin does not contain data for the tree's vertex buffer with the same index.

This function essentially allows the tree to say to the leaf bin, "Give me your index data and render batches for a specific vertex buffer index".

```
CLeafBinData * CLeafBin::GetBinData( unsigned long nIndex ) const
{
    // Validate parameters
    if ( nIndex >= m_nVBCount ) return NULL;

    // Retrieve the item
    return m_ppBinData[ nIndex ];
}
```

As you can see, as long as the passed index is within the bounds of the array (i.e., less than m_nVBCount), the pointer contained in the passed index of the CLeafBinData array is returned.

## AddVisibleData – CLeafBin

This method should be familiar since we looked at various versions of this type of function as a means to initially understand the render batch system.

We saw earlier that it is called from the CBaseLeaf::SetVisible method. When a leaf is deemed to be visible during the update pass, a loop is made through the RenderData items stored in that leaf (one exists for each subset in the leaf). An inner loop was then used to iterate through the (possible) multiple Elements stored in that RenderData item. There was an Element in RenderData structure for each vertex buffer that contains geometry from that leaf for the given subset. Each Element describes a run of triangles in that leaf that use the subset along with the index of the vertex buffer in which the run of triangles is contained. The AddVisibleData method of CLeafBin is then called and passed the Element information: the vertex buffer index in which the run is stored, the start index in the leaf bin's index buffer associated with that vertex buffer, and the number of triangles in the run.

```
void CLeafBin::AddVisibleData( unsigned char VBIndex,
                               unsigned long IndexStart,
                               unsigned long PrimitiveCount )
{
```

The vertex buffer index of the render element information passed in also describes the index of the CLeafBinData structure in the leaf bin's array of the index buffer and render batch list associated with this vertex buffer. Therefore, we fetch the pointer to the relevant CLeafBinData structure from the bin's array.

```
    CLeafBinData * pData = m_ppBinData[ VBIndex ];
    if ( !pData ) return;
```

We then fetch the values of the CLeafBinData item's m_nLastIndexStart and m_nLastPrimitiveCount members into local variables. m_nLastIndexStart will currently contain the first index of the first triangle in the current render batch we are trying to build for this index buffer. The m_nLastPrimitiveCount member will contain the number of adjacent triangles we have been able to pack into the batch so far. Hopefully, the range of triangles passed in will follow on exactly where m_nLastPrimitiveCount ends in the index buffer, allowing us to add the new triangle range to the current batch being compiled.

```
    ULONG LastIndexStart      = pData->m_nLastIndexStart,
          LastPrimitiveCount  = pData->m_nLastPrimitiveCount;
```

If LastPrimitiveCount equals zero, it means that we have not yet collected any triangles for the current batch and therefore the triangle range passed in should be used to start the beginning of a new render batch. When this is the case, we simply copy the passed IndexStart and PrimitiveCount into the LastIndexStart and LastPrimitiveCount variables. Hopefully, the next time this function is called for another visible leaf, we can add the passed triangle range to the end of this batch we have just started.

```
    // Build up batch lists
```

```
    if ( LastPrimitiveCount == 0 )
    {
        // We don't have any data yet so just store initial values
        LastIndexStart     = IndexStart;
        LastPrimitiveCount = PrimitiveCount;

    } // End if no data
```

If this is not the case then we must be currently in the middle of compiling a batch of adjacent triangles for the CLeafBinData's index buffer. Therefore, we test to see if the start index passed into the function follows on immediately after the current batch we are compiling ends (LastIndexStart+(LastPrimitiveCount*3)). If it does, then the passed triangles continue the adjacent run of triangles we have already recorded and we can just add them to the current batch by increasing the primitive count to compensate for the number of primitives passed into the function.

```
    else if ( IndexStart == (LastIndexStart + (LastPrimitiveCount * 3)) )
    {
        // Just grow the primitive count
        LastPrimitiveCount += PrimitiveCount;

    } // End if consecutive primitives
```

If none of the above cases are true then it means that we are in the middle of compiling a render batch but the triangle(s) we have just been passed do not follow on continuously from the current batch we are compiling. This essentially ends the current batch we are compiling and the new triangles we have just been passed will be used to start a new batch collection process.

```
   else
   {
   // Store any previous data for rendering
   pData->m_pRenderBatches[pData->m_nBatchCount].IndexStart= LastIndexStart;
   pData->m_pRenderBatches[pData->m_nBatchCount].PrimitiveCount=LastPrimitiveCount;
   pData->m_nBatchCount++;

   // Start the new list
   LastIndexStart     = IndexStart;
   LastPrimitiveCount = PrimitiveCount;

   } // End if new batch
```

As you can see, because LastIndexStart and LastPrimitiveCount describe the block of adjacent triangles we managed to collect until this function was called, we store them in the CLeafBinData object's render batch array and increase its render batch count. This render batch is now finished and we need to start a new collection process. We do this by assigning the start index and primitive count of the range of triangles passed in to the LastIndexStart and LastPrimitiveCount local variables which will be used from this point on. Finally we store the local LastIndexStart and LastPrimitiveCount values in the CLeafBinData member variables so that they are available the next time this method is called.

```
    // Store the updated values
    pData->m_nLastIndexStart     = LastIndexStart;
```

```
    pData->m_nLastPrimitiveCount = LastPrimitiveCount;
}
```

## Render - CLeafBin

The CLeafBin::Render method is called by the tree to render all the triangles contained in the leaf bin. This method will not typically be called by the application, as it will use the CBaseTree::DrawSubset method for each subset that it wishes to render instead. DrawSubset simply issues the render call to the leaf bin with the matching attribute.

The Render function is shown below. It essentially loops through each pointer in the leaf bin's CLeafBinData array to process and render the data contained in each.

```
void CLeafBin::Render( LPDIRECT3DDEVICE9 pDevice )
{
    ULONG i, j;

    // Loop through each vertex buffer
    for ( i = 0; i < m_nVBCount; ++i )
    {
        CLeafBinData * pData = m_ppBinData[ i ];
        if ( !pData ) continue;
```

As you can see in the above code, the current iteration of the loop is skipped if this leaf bin does not have a CLeafBinData item for a given vertex buffer.

The next function call requires some explanation. You will recall that in the AddVisibleData method we used the m_nLastIndexStart and m_nLastPrimitiveCount members of the CLeafBinData item to record the start and triangle count of the batch of triangles being compiled. Only when we are passed triangles that do not continue the current batch is the batch data copied into a RenderBatch data structure. This means that at the end of the visibility process, we may be in the middle of compiling a batch of triangles that have not yet been committed to the render batch list. For example, consider the last leaf processed for a given subset. Even if it starts a branch new batch, because no further triangles are passed, the AddVisibleData function is never passed triangles that break the batch, and therefore it is never aware that the batch ever ended. This means that the m_nLastIndexStart and m_nLastPrimitiveCount members of each CLeafBinData structure describe their final batches, but they are not yet committed to a render batch structure. Therefore, before we draw the CLeafBinData render batches, we must call AddVisibleData for this item one last time, passing in both an index start and primitive count of zero.

```
        // Commit any last piece of data to the render queue and reset "Last"
        // values back to 0
        AddVisibleData( (unsigned char)i, 0, 0 );
```

As the first parameter, we pass the current vertex buffer index we are processing which describes (to the AddVisibleData method) which CLeafBinData object we are adding triangles for. Also, because we pass 0 in as the IndexStart parameter, which could not possibly continue any batch that has been compiled,

the function will recognize that the last batch has ended and will store it in the CLeafBinData object's render batch list. Finally, this will cause the function to start a new batch to be compiled with an index start and primitive count set to the values passed in. Since we pass zero for both of these values, this essentially resets the CLeafBinData's m_nLastIndexStart and m_nLastPrimitiveCount members back to zero for the next visibility pass.

At this point, we have the CLeafBinData pointer, so we send its vertex and index buffers to the device and set the FVF for the vertex type being used.

```
        // Set the stream sources to the device
        pDevice->SetStreamSource( 0, pData->m_pVertexBuffer, 0, sizeof(CVertex) );
        pDevice->SetIndices( pData->m_pIndexBuffer );

        // Set the FVF
        pDevice->SetFVF( VERTEX_FVF );
```

Finally, we loop through every RenderBatch structure stored in the CLeafBinData's render batch list and render the triangles described by each.

```
        // Render the leaves
        for ( j = 0; j < pData->m_nBatchCount; ++j )
        {
            CLeafBinData::RenderBatch & Batch = pData->m_pRenderBatches[j];

            // Render any data
            pDevice->DrawIndexedPrimitive( D3DPT_TRIANGLELIST,
                                           0,
                                           0,
                                           pData->m_nVertexCount,
                                           Batch.IndexStart,
                                           Batch.PrimitiveCount );
        } // Next element
    } // Next Vertex Buffer
}
```

We have now covered all of the rendering code for CBaseLeaf, CLeafBin, and CLeafBinData and you should have a fairly good understanding of how these components work. We will now cover the new rendering code in the top level object (CBaseTree) that brings all of these components together under one system.

# 15.4 CBaseTree – Adding Rendering Support

Adding the code that actually performs the visibility pass and renders the data in the tree is relatively trivial. The CBaseTree methods that perform these tasks will be small and simple as they just pass the requests on to the leaf bins. As we have seen, it is the leaf bins that contain the triangles and have knowledge of how to render them.

The majority of work we will have to do in CBaseTree is the implementation of the BuildRenderData method. As you will soon see, this method is responsible for building all the vertex buffers used by the tree, allocating the leaf bins for each attribute used by the tree, building the index buffers for each leaf bin, and a host of other tasks. Most of our discussion of CBaseTree will be spent describing how this function works. If you have already looked at the source code to this function, you might have noted that it seems a little intimidating. Do not worry though; we will break this function down (and the functions it calls) piece by piece so that we understand everything that is going on when building the render data.

## 15.4.1 CBaseTree – The Source Code

Surprisingly few methods and members have to be added to CBaseTree in order to implement the rendering system. The upgraded class declaration is shown below. The new methods and members that pertain to the rendering system are highlighted in bold. This declaration will be followed by a detailed description of the new members and an examination of each new method.

```
class CBaseTree : public ISpatialTree
{
public:

    // Friend list for CBaseTree
    friend void CBaseLeaf::SetVisible( bool bVisible );

    // Constructors & Destructors for This Class.
    virtual ~CBaseTree( );
             CBaseTree( LPDIRECT3DDEVICE9 pDevice, bool bHardwareTnL );


    // Public Virtual Functions for This Class (from base).
    virtual bool            AddPolygon        ( CPolygon * pPolygon );
    virtual bool            AddDetailArea     ( const TreeDetailArea &DetailArea );

    virtual bool            Repair              ( );

    virtual PolygonList    &GetPolygonList      ( );
    virtual DetailAreaList &GetDetailAreaList   ( );
    virtual LeafList       &GetLeafList         ( );
    virtual LeafList       &GetVisibleLeafList  ( );

    virtual void            DrawSubset          ( unsigned long nAttribID );
    virtual void            ProcessVisibility   ( CCamera & Camera );


    // Public Functions for This Class
    CLeafBin *              GetLeafBin          ( unsigned long nAttribID );
    bool                    AddLeaf             ( CBaseLeaf * pLeaf );
    bool                    AddVisibleLeaf      ( CBaseLeaf * pLeaf );
    bool                    BuildRenderData     ( );

protected:
```

```
    // STL Typedefs
    typedef std::map<ULONG,CLeafBin*>       LeafBinMap;

    // Protected Functions for This Class.
public:
    void   DrawBoundingBox     ( const D3DXVECTOR3 & Min,
                                 const D3DXVECTOR3 & Max,
                                       ULONG Color,
                                       bool bZEnable = false );

protected:
    void   DrawScreenTint      ( ULONG Color );
    void                    CalculatePolyBounds ( );

    void                    RepairTJunctions    ( CPolygon * pPoly1,
                                                  CPolygon * pPoly2 );

    ULONG                   WeldBuffers ( ULONG VertexCount,
                                          CVertex * pVertices,
                                          std::map<ULONG,ULONG*> & BinIndexData,
                                          std::map<ULONG,ULONG> & BinSizes,
                                          ULONG * pFirstVertex = NULL,
                                          ULONG * pPreviousVertex = NULL );

    bool                    CommitBuffers( ULONG VertexCount,
                                           CVertex * pVertices,
                                           std::map<ULONG,ULONG*> & BinIndexData,
                                           std::map<ULONG,ULONG> & BinSizes,
                                           bool b32BitIndices );

    bool                    PostBuild           (  );


    // Protected Variables for This Class.
    LPDIRECT3DDEVICE9           m_pD3DDevice;
    LPDIRECT3DVERTEXBUFFER9    *m_ppVertexBuffer;
    unsigned char               m_nVBCount;
    bool                        m_bHardwareTnL;

    LeafBinMap                  m_LeafBins;

    LeafList                    m_Leaves;
    PolygonList                 m_Polygons;
    DetailAreaList              m_DetailAreas;

    LeafList                    m_VisibleLeaves;
};
```

Let us discuss a new type definition we have in this namespace:

```
// STL Typedefs
   typedef std::map<ULONG,CLeafBin*>  LeafBinMap;
```

We will store the tree's leaf bins in an STL map. You will recall from our C++ programming course (Module II) that an STL map is basically a hash table that allows us to store key/value pairs in a manner that makes it very efficient to look up that value by specifying the key. We type define LeafBinMap to be an STL map that stores a ULONG as the key and a CLeafBin pointer as the value paired with that key. The key will by the attribute ID of the leaf bin pointed to by the value assigned to that key. Therefore, the tree would be able to very efficiently fetch the leaf bin for subset 6's triangles (for example) by doing something like this:

```
CLeafBin = LeafBinMap [ 6 ];
```

This looks like a simple array look up, so why do we not just use an array to store the leaf bin pointers where the attribute ID is the index into the array where the associated leaf bin is stored?

What we must remember is that our scene uses global attribute IDs and that the static geometry compiled in the tree may only use some of these. For example, assume that after loading our scene and all of our dynamic objects that the scene has created 120 attributes. Imagine however, that the static polygons in the spatial tree only use subsets 6, 40, and 120 out of the 120 possible scene attributes. The leaf bin array would need to be 120 elements in size when only 3 of the elements would contain valid pointers to leaf bins. That is a waste of memory. Alternatively, the map would only have three rows in its hash table. Of course, you could use an array that is the exact size of the number of attributes used by the tree but that would mean the array indices no longer match the attribute IDs of the leaf bins stored there. In this case, whenever you need to fetch the leaf bin pointer for a given subset, you would have to loop through the array searching for the leaf bin with the attribute ID you are looking for. This will hamper performance when the tree contains a lot of attributes (which would have to be searched through every time the DrawSubset method is called). This is especially true if the tree data requires multiple render passes. A map is definitely a better choice of data structure here since it conserves memory and performs efficient value lookups.

Let is now take a look at the new member variables in CBaseTree that have been added to support the render system.

**LPDIRECT3DDEVICE9 m_pD3DDevice**
This pointer is passed into the constructor of CBaseTree (called by the constructor of the derived classes) and represents the device which the tree will use to render. If this is set to NULL, the BuildRenderData method will perform no action and will return immediately. This is very useful if you intend to use the tree only for collision queries. Passing NULL in as this parameter to the constructor of a derived class allows you to prevent the render system from being activated and avoid the unnecessary memory consumption. No vertex buffers, index buffers, or leaf bins will be generated in this instance. By setting this to point at a valid device, we inform CBaseTree that we intend to use its render system.

**LPDIRECT3DVERTEXBUFFER9 *m_ppVertexBuffer**
This will be set to NULL until CBaseTree::BuildRenderData executes. At the end of the renderable data building process is will point to an array of vertex buffer pointers. That is, if it was determined during the building of the render data that the static scene needs to be contained in three vertex buffers, this will point to an array of three vertex buffer pointers currently containing the vertex data of the tree's static geometry.

**unsigned char m_nVBCount**
After the render data building process has been completed, this member will contain the number of vertex buffers being used by the render system and the number of elements in the above vertex buffer pointer array.

**bool m_bHardwareTnL**
This Boolean is also passed into the constructor of CBaseTree (called from the derived classes) and tells the render system whether the graphics hardware on the current system on which the application is running supports transformation and lighting of vertices in hardware. This Boolean will ultimately determine the vertex buffer creation flags that need to be passed to DirectX when the tree needs to create a new vertex buffer.

**LeafBinMap m_LeafBins**
This is an STL map containing the leaf bin pointers used by the tree. This map will be empty until CBaseTree::BuildRenderData is called. As the leaf bins are created, their pointers are added to this map. The key for each row in the table is the attribute ID of the leaf bin. The value assigned to that key will be the leaf bin pointer.

**LeafList m_VisibleLeaves**
We discussed this new leaf list when we discussed the CBaseLeaf source code. This leaf list is emptied just before a new visibility determination pass is performed on the tree. As each visible leaf is encountered, its pointer is added to this list. We saw the code that added the leaf pointers to this list in the CBaseLeaf::SetVisible method earlier in the lesson. At the end of each ISpatialTree::ProcessVisibility call, this list will contain the pointers for all the leaves that are currently deemed visible by the render system (i.e., leaves that are currently inside the camera frustum).

We will now discuss the new methods in CBaseTree that pertain to the rendering system.

## AddVisibleLeaf – CBaseTree

We saw this method being called earlier in the lesson when we examined the code to CBaseLeaf::SetVisible. It is passed the pointer of a leaf which it then adds to the tree's visible leaf list. Remembering that LeafList is just an STL list, we can see that this function simply adds the passed pointer to the end of the list.

```
bool CBaseTree::AddVisibleLeaf( CBaseLeaf * pLeaf )
{
    try
    {
        // Add to the leaf list
        m_VisibleLeaves.push_back( pLeaf );

    } // End Try Block

    catch ( ... )
```

```
    {
        return false;

    } // End Catch Block

    // Success!
    return true;
}
```

The application should never call this function; it should be left to the tree's visibility system to build this list with each visibility pass. In particular, it is only the CBaseLeaf::SetVisible method that ever calls this function to add its own pointer to the list.

## GetVisibleLeafList - CBaseTree

This method is exposed purely for the application's benefit. The application can call this method to get returned the list of currently visible leaves. This is the leaf list that was compiled the last time the application called the ISpatialTree::ProcessVisibility method.

```
CBaseTree::LeafList & CBaseTree::GetVisibleLeafList()
{
    return m_VisibleLeaves;
}
```

While it might not seem apparent just yet why the application may want to know which leaves are visible (especially when the tree takes care of rendering its visible polygon data), we will see later how this information will be important for the efficient rendering of dynamic objects. The application is going to be responsible for rendering these objects (not the tree) and thus, retrieving the list of currently visible leaves from the tree and then retrieving the dynamic object pointers stored in those leaves, provides the application with the list of currently visible dynamic objects that need to be rendered.

## GetLeafBin - CBaseTree

The GetLeafBin method will generally never be called by the application. It is used by the tree during the building of the render data to fetch a leaf bin pointer from the map. For example, if the build process finds a polygon with an attribute of 7 it knows that it has to add its indices to leaf bin 7. As this leaf bin might have already been created (when processing another polygon with the same attribute ID), this method will quickly retrieve the pointer to that leaf bin and return it to the building process. If there is currently no key in the leaf bin map that matches the passed attribute ID the function returns NULL. This informs the calling function that the leaf bin associated with subset 7 has not yet been created, so it should do so now.

```
CLeafBin * CBaseTree::GetLeafBin( unsigned long nAttribID )
{
    LeafBinMap::iterator Item = m_LeafBins.find( nAttribID );
```

```
    // We don't store this attrib ID
    if ( Item == m_LeafBins.end() ) return NULL;

    // Return the actual item
    return Item->second;
}
```

Although it might seem strange that we have not just indexed into the map using the attribute ID (using an array style approach), if we do this and the key/value pair does not exist in the table, this new row will be added with a value of NULL. However, we want this function to return the pointer only if it does exists and not create the row in the hash table if it does not. For this reason, we use the STL map's find method.

The find method does not perform a search through the entire table row by row as its name would perhaps imply. It efficiently jumps straight to the row in the table with the matching key and returns the iterator to that item in the table. If the key (attribute ID) did not exist in the map, then the iterator will point to the end of the map (like a hash table's equivalent of a NULL return). If this is the case, we return NULL from the function. Otherwise we fetch the leaf bin pointer stored in the returned iterator's 'second' member and return it (the key itself is stored in the 'first' member of the iterator).

We have now looked at the small utility functions in CBaseTree which we will need during the render data building process. In the next section, we will discuss the new methods in CBaseTree that are executed once, just after the tree has been built, to compile the tree's static polygon data into vertex and index buffers, and leaf bins.

## 15.4.2 The Building Phase – Compiling the Render Data

The entire building phase is invoked from the CBaseTree::PostBuild method introduced in the previous lesson. This method is called by the derived classes just after the tree has been built (at the very bottom of their Build functions). We discussed how and why this function calls the CalculatePolyBounds function in the previous lesson to build and store the AABBs for each CPolygon in the tree. What we have not yet seen are the contents of the second method that it calls. The BuildRenderData method is the method that compiles the data into a renderable format. It creates all the vertex and index buffers used by the tree and creates and populates the leaf bins.

```
bool CBaseTree::PostBuild(  )
{
    // Calculate the polygon bounding boxes
    CalculatePolyBounds( );

    // Build the render data
    return BuildRenderData( );
}
```

## BuildRenderData - CBaseTree

This function is a very large and seemingly complex function. The function could be made shorter by moving multiple processes into a single loop (instead of doing a separate loop for each process), but this would have made the function must harder to break into components for the purpose of explanation. Before we dive into the code, we will discuss what purpose of this function is, the tasks it must perform, and the way those tasks will be implemented. We will then find that this code is not nearly as intimidating as it first seems.

When the function is first called, no vertex buffers or index buffers will have been created. The function will allocate a temporary array of vertices that will be used to collect the vertices of each polygon it processes. We can think of this array initially as collecting the vertices for the first (and possibly only) vertex buffer that will be created. Likewise, the function will also allocate an array of temporary buffers to create the indices for each leaf bin. That is, if the scene contains polygon data for 15 subsets, we will create an array to store 15 sets of indices, one for each leaf bin. We can think of these initial index arrays as being used to collect the indices that are associated with the first vertex buffer.

The thing the function will do is call the CBaseTree::Repair method. We discussed this function in the previous lesson and saw how removes T-junctions in the polygon dataset. We call this automatically in the case of this particular function because we know this data is being used for rendering and we certainly do not want any T-junctions in our renderable data (they cause lighting artifacts and generate sparklies). After the Repair function returns, we have our data ready to be compiled for rendering with all T-junctions removed.

The next step will be to call the CollectLeavesAABB method passing in the bounding box of the root node. This will collect all the leaves of the tree into a leaf list in the exact order in which the tree would be traversed if every leaf was visible. This is very important as it allows us to add the triangles from each leaf into the leaf bin index buffers in the order in which they will be traversed. This greatly increases our changes of creating large render batches during the visibility pass.

Once we have the leaf list, we will loop through and process each leaf. For each leaf we will process the polygons in that leaf. For each polygon we will retrieve its attribute ID and fetch the associated leaf bin for that subset. If a leaf bin has not yet been created for that subset, we will create a new leaf bin for it. The whole point of this process is to make sure that at the end of the loop we have a leaf bin created and stored in the tree for every subset used by the tree. As we process each polygon, we will also calculate the number of indices that should be generated for it and add it to the temporary index array for that leaf bin. Therefore, at the end of the leaf loop we will have made sure that a leaf bin has been created for every subset used by the tree, and we will have recorded exactly how many indices we will need to allocate the temporary index arrays (for each leaf bin) to hold.

Before we go any further, let us cover the first sections of the code that perform these steps. First we will look at the local variables allocated at the top of the function as some of them are extremely significant.

```
bool CBaseTree::BuildRenderData(  )
{
```

```
    LeafList::iterator LeafIterator;
    map<ULONG,ULONG>   BinSizes;
    map<ULONG,ULONG*>  BinIndexData;
    LeafList           Leaves;
    ULONG              nMaxVertexCount, nPolygonCount, nTotalVertices;
    ULONG              nVertexCount, i, j;
    D3DCAPS9           Caps;
    CBaseLeaf        * pLeaf;
    CLeafBin         * pLeafBin;
    CVertex          * pVertices = NULL;
    bool               b32BitIndices = false;
    D3DXVECTOR3        Min, Max;

    CBaseLeaf::RenderData          * pRenderData;
    CBaseLeaf::RenderData::Element * pElement;
```

Notice the two local variables that are highlighted in bold in the above code. These two STL maps are vital to the building process. The first stores ULONGs as both its keys and values and will be used in the initial loop (just described) to record the number of indices we found for each subset/leaf bin. The key of each row in the hash table will be the attribute ID and the value will be the number of indices we recorded that will need to be generated for it. The second STL map is the partner of the previous one and will be used to store the temporary index arrays being collected for each leaf bin. The key is a ULONG which once again stores the attribute ID that its value is relative to. The value is a ULONG pointer that will be allocated to store as many indices as is described by the row in the BinSizes map with the matching key (attribute ID).

For example, imagine that after looping through the leaf list and processing however many indices will be generated by all the polygons, we find that the tree uses subsets 3, 6, and 9. Let us also assume that the number of indices recorded for each subset was 100, 200, and 300, respectively. At the end of the first loop, the following key/value pairs will be stored in the BinSizes map.

BinSizes [ 3 ] = 100
BinSizes [ 6 ] = 200
BinSizes [ 9 ] = 300

The three rows in the map now tell us that we will need to create a temporary index array to collect the 100 indices for the first leaf bin (subset 3), an array to collect 200 indices for the second leaf bin (subset 6) and an array of 300 to collect the indices for the third leaf bin (subset 9).

After the loop, we can then allocate these temporary arrays in the BinIndexData map with code like the following:

```
// Allocate temporary buffers for each of the bin items
map<ULONG,ULONG>::iterator BinSizeIterator = BinSizes.begin();
for ( ; BinSizeIterator != BinSizes.end(); ++BinSizeIterator )
{
    ULONG AttribID = BinSizeIterator->first;
    ULONG Size     = BinSizeIterator->second;

    if ( Size > 0 )
```

```
            BinIndexData[ AttribID ] = new ULONG[ Size ];
        else
            BinIndexData[ AttribID ] = NULL;

        // Reset sizes back to 0, we're going to tally as we go
        BinSizes[ AttribID ] = 0;

    } // Next Bin
```

As you can see, we loop through each row in the BinSizes table (one per subset used by the tree) and extract the key (attribute ID) and the indices size. We then use this size to allocate an array of indices which is assigned to the row in the BinIndexData map with a matching attribute. The function would then proceed to fill up these index arrays with the indices intended for each leaf bin.

Now that we know how these two maps will be used in tandem to represent the index arrays being compiled for each leaf bin, let us examine the first section of the function that creates the leaf bins and records the number of indices that will be needed for each subset.

The first thing we do is test to see if the device pointer is valid. If not, there is no point in initializing the render system and we simply return. Also, if the tree has no leaves we also have no polygon data so we return false. Provided the device and the leaf array are valid, we then call the CBaseTree::Repair method to remove any T-junctions that may exist in the polygon data.

```
    // Instructed not to build data?
    if ( !m_pD3DDevice ) return true;

    // Anything to do ?
    if ( m_Leaves.size() == 0 ) return false;

    // First thing we need to do is repair any T-Junctions created during the build
    Repair( );
```

We will now need to fetch the list of leaves contained in the tree. However, as this leaf list will be iterated through in order and each leaf's polygons added to the leaf bins, we want this leaf list ordered in the same way that the leaves will be encountered when the visibility traversal is performed. This will ensure that we get the most optimal render batches during visibility traversal. To do this we simply fetch the bounding box of the scene (i.e., the root node's bounding box) and feed this box into the CollectLeavesAABB method. When the function returns, the local leaf list passed in as the first parameter will be filled with all the leaves contained in the tree in traversal order.

```
    // Retrieve the leaf list in TRAVERSAL order to ensure that the
    // render batching works as efficiently as possible
    GetSceneBounds( Min, Max );
    CollectLeavesAABB( Leaves, Min, Max );
```

We will now set up a loop that will loop through each leaf and record the total number of indices that will be generated for each subset. We will also allocate the leaf bins for each subset.

For each leaf we fetch its polygon count and then loop through each polygon in the leaf. Notice also that at the top of the next section of code, we set the local variable nTotalVertices to zero. This will be used to also record the total number of vertices stored in the tree (from all polygons).

```
    // Iterate through all of the polygons in the tree and allocate/add to
    // leaf bins.  In addition, total up the index counts required for each bin.
    nTotalVertices = 0;

    for ( LeafIterator = Leaves.begin();
          LeafIterator != Leaves.end();
          ++LeafIterator )
    {
        // Retrieve leaf
        pLeaf = (CBaseLeaf*)(*LeafIterator);
        if ( !pLeaf ) continue;

        // Loop through each of the polygons stored in this leaf
        nPolygonCount = pLeaf->GetPolygonCount();

        for ( i = 0; i < nPolygonCount; ++i )
        {
```

In this loop we fetch a pointer to each polygon in the leaf. If for some reason its vertex count is smaller than 3, then this is an invalid triangle and we skip it. We also skip the polygon if its m_bVisible Boolean is set to false. The application can set a CPolygon's visibility status to false prior to registering it with the tree. This will allow the application to register polygons that will still be used for collision detection queries but will not be rendered. You will see later when we discuss the changes to our application that we set this flag to false for transparent polygons before they are registered with the tree. Transparent polygons will be rendered by our BSP tree (discussed in the next chapter) so that we can get perfect back to front ordering. However, we still want to register such polygons with the spatial tree so that a 'Glass' polygon for example is still considered a solid polygon by the collision system. Therefore, this flag allows our application to say to the tree, "Here is a static polygon for collision purposes, but do not render it as I will render it on my own using a different methodology."

```
            CPolygon * pPoly = pLeaf->GetPolygon( i );

            // Skip if it's invalid
            if ( !pPoly || pPoly->m_nVertexCount < 3 ) continue;

            // Skip if it's not visible
            if ( !pPoly->m_bVisible ) continue;

            // Retrieve the leaf bin for this attribute
            pLeafBin = GetLeafBin( pPoly->m_nAttribID );
```

In the code above, we see that once we have a pointer to a valid renderable CPolygon, we pass its attribute ID into the CBaseTree::GetLeafBin function. If a leaf bin has already been created for this subset, a pointer to the leaf bin will be returned and stored in the pLeafBin local variable pointer. If we have encountered a polygon which belongs to a subset that we have not yet encountered and created a

leaf bin for, the GetLeafBin method will find no such leaf bin and the returned value of NULL will be stored in pLeafBin.

If pLeafBin is NULL then it means we have found a polygon that a leaf bin does not yet exist for, so we had better create one. In the next section of code, we allocate a new leaf bin (passing in the attribute ID the leaf bin is intended to store triangles for and the total leaf size of the tree) and store the leaf bin's pointer in the CBaseTree's m_LeafBins map with a key equal to the attribute ID.

```
            // Determine if there is a matching bin already in existance
            if ( !pLeafBin )
            {
                try
                {
                    // Allocate a new leaf bin
                    pLeafBin = new CLeafBin( pPoly->m_nAttribID, m_Leaves.size() );
                    if ( !pLeafBin ) throw std::bad_alloc(); // VC++ Compat

                    // Add to the leaf bin list
                    m_LeafBins[ pPoly->m_nAttribID ] = pLeafBin;

                } // End Try Block

                catch ( ... )
                {
                    // Clean up and fail
                    if ( pLeafBin ) delete pLeafBin;
                    return false;

                } // End Catch Block

            } // End if no bin existing
```

At this point we have either confirmed that a bin exists that will accept this polygon or we have created a new one that will collect polygons of this type. We will now increment the value stored in the row of the BinSizes map that has a key equal to the current polygon's subset. This allows us to record the indices for each subset in the BinSizes map as we go. We will need to break our clockwise winding polygons into a series of triangles to store in a triangle list (for the index buffer), and we have previously discovered that the number of triangles produced will be equal to the number of vertices in the polygon minus 2 (see Figure 15.3).

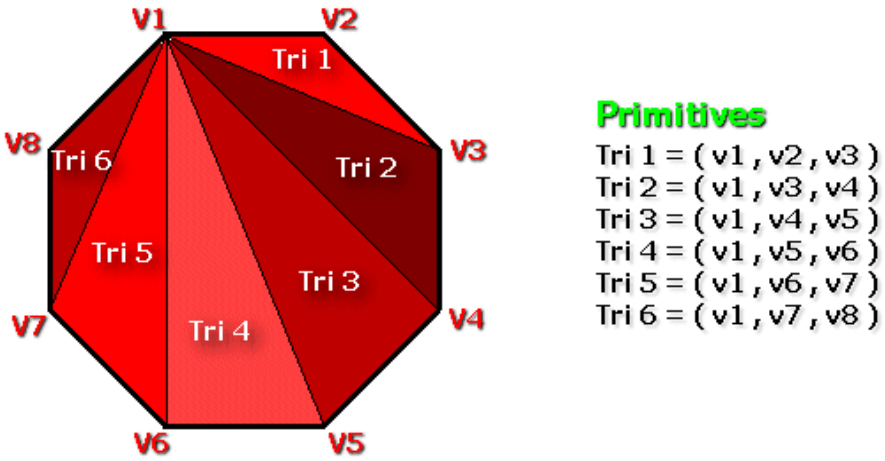



**Figure 15.3**

In this example we can see that the 8 vertices describe 6 triangles (vertex count - 2). Therefore, as each triangle will be described by three indices in the index buffer, the total number of indices that will be generated by the current polygon will be (VertexCount - 2) * 3.

In the following code you can see that we perform this calculation and add the resulting index count to the value of the row in the map with a key that matches the attribute ID of the polygon. We also add the vertex count of the polygon to the nTotalVertices local variable so that we record the total number of vertices in the entire tree after the entire leaf loop completes.

```
            // Add each of the triangles to the total index count.
            BinSizes[ pPoly->m_nAttribID ] += (pPoly->m_nVertexCount - 2) * 3;

            // Total up vertices
            nTotalVertices += pPoly->m_nVertexCount;

        } // Next Polygon
    } // Next Leaf
```

At this point we are no longer in any loops and we have created the leaf bins for every subset in the tree. We have also recorded the total number of vertices stored in the tree in the nTotalVertices variable. Finally, we have recorded, for each subset, the total number of indices that will be generated for that subset. These index counts are stored in the BinSizes map, keyed by subset/attribute ID. At this point we have collected no vertices or indices and we have not yet added any data to any leaf bins.

What we know is how many vertices we are going to need to store, which allows us to find out two things. First, if the vertex count is larger than the maximum number of vertices the device allows to be stored in one vertex buffer, we know we will need multiple vertex buffers. Second, if the number of vertices we have exceeds 65,535 (0xffff), we know that we will need 32-bit indices to reference them as we cannot fit values greater than this size into a 16-bit index.

First we get the capabilities of the device and retrieve the value of the maximum vertex index capability. This is stored in the D3DCAPS9::MaxVertexIndex member. As the maximum vertex index tells us the maximum index of a vertex that is allowed on the current device, adding one to this amount (remembering that indices are zero based) tells us the total number of vertices that can be placed in a single vertex buffer on this device. We store this value in the nMaxVertexCount local variable.

```
    // Calculate the maximum vertex count the card is capable of storing
    m_pD3DDevice->GetDeviceCaps( &Caps );
    nMaxVertexCount = Caps.MaxVertexIndex + 1;
```

Now that we know the maximum number of vertices that can be stored in a vertex buffer, we should test to see if this is smaller than the total number of vertices used by our tree's static geometry. If it is not, then it means the current device can handle all the vertices we have in a single vertex buffer. When this is the case we simply modify the value of nMaxVertexCount to equal the total number of vertices in the scene. We can think of the nMaxVertexCount member as containing the vertex count at which we will have to create a new vertex buffer during the vertex collection process.

```
    // If it's more than capable, just clamp it to the scene vert count
    if ( nMaxVertexCount > nTotalVertices ) nMaxVertexCount = nTotalVertices;
```

At this point, nMaxVertexCount will contain either the total number of vertices in our tree (if the hardware can handle them all) or the maximum number of vertices that can fit in a single vertex buffer on this device. If this is larger than 65,535 (0xffff) then we will need to create 32-bit index buffers, so we set the local b32BitIndices Boolean to true so we will know this when creating leaf bin index buffers later.

```
    // Determine if we need to use 32 bit indices or not
    b32BitIndices = ((nMaxVertexCount - 1) > 0xFFFF) ? true : false;
```

Since nMaxVertexCount describes the maximum size that the vertex buffers we create will be, we only create a vertex buffer once we have collected this many vertices. Therefore, we create a temporary vertex array of this size that will be used to collect the vertices for each vertex buffer. Once this buffer is full, a hardware vertex buffer will be created, the data copied over, and the array's count set back to zero again. The array will then be reused to collect vertices for the next vertex buffer, and so on.

```
    // Allocate temporary vertex array
    pVertices = new CVertex[nMaxVertexCount];
    if ( !pVertices ) return false;

    nVertexCount = 0;
```

The BinSizes map contains the total number of indices that will be generated for each leaf bin's index buffers. Just as with the temporary vertex array, we will use the BinIndexData map to store pointers to temporary index buffers that will be used to collect the indices for each leaf bin for the current vertex buffer. Once the pVertices array is full and the vertices are committed to a vertex buffer, index buffers will be created for each subset that we collected indices for and the contents of these temporary index arrays will be copied into the index buffers. Each new index buffer will then be added to a leaf bin and

the temporary index arrays stored in the BinIndexData map will be reused to collect the indices for each subset for the next vertex buffer, and so on.

The next section of code loops through every row in the BinSizes map and fetches each key/value pair into the AttribID and Size local variables. If the size of a given row is greater than zero then it means that there are some indices that will be generated for this leaf bin during the index collection process (shown in a moment). Therefore, we allocate an array of indices for each leaf bin used by the scene that is large enough in size to store all the indices recorded for that leaf bin in the above code. After this loop is finished, the LeafIndexData map will store a pointer to an index array for each leaf bin.

```
    // Allocate temporary buffers for each of the bin items
    map<ULONG,ULONG>::iterator BinSizeIterator = BinSizes.begin();
    for ( ; BinSizeIterator != BinSizes.end(); ++BinSizeIterator )
    {
        ULONG AttribID = BinSizeIterator->first;
        ULONG Size     = BinSizeIterator->second;

        if ( Size > 0 )
            BinIndexData[ AttribID ] = new ULONG[ Size ];
        else
            BinIndexData[ AttribID ] = NULL;

        // Reset sizes back to 0, we're going to tally as we go
        BinSizes[ AttribID ] = 0;

    } // Next Bin
```

Notice that the BinSizes map which had its values recorded in the first loop has only been used for the allocation of the temporary index arrays for each subset. As we allocate the index array for each leaf bin, we reset its associated BinSize value to zero. That is because we will use the BinSize map later to record how many indices we have currently added to each leaf bin, which will be used in the generation of the leaf RenderData structures (more on this in a moment).

In the next section of code we will loop through the leaf list again, and through each leaf's polygon list and actually collect the indices and vertices of the polygons into their respective arrays. We will also create and populate the RenderData structures at each leaf (one for each subset contained in that leaf) and create the Elements for each RenderData structure. Remember, each Element in a RenderData structure contains a triangle range (index start and primitive count) for the current vertex buffer being compiled. If all the vertices fit in a single buffer, this loop will only ever create one render element in each leaf's RenderData structure(s). We shall also see two calls to CBaseTree member functions we have not yet covered. The first is called CBaseTree::CommitBuffers and is called once the vertex buffers are filled. It is this function which takes the vertices collected in the temporary vertex array and copies them into a newly allocated vertex buffer. It is also this function that copies the data stored in the index arrays for each subset and copies them into newly allocated index buffers. These index buffers are then stored in newly allocated CLeafBinData structures and added to the leaf bins. We will look at the code to this function in a moment. For now just know that this function is called when we have collected enough vertices and need to create actual Direct3D vertex and index buffers. It is called CommitBuffers because it is this function that takes the data stored in the temporary vertex and index arrays and commits them to static vertex and index buffers. After the CommitBuffers function returns, the data in

the temporary vertex and index arrays are discarded and the current vertex buffer count will have been increased from zero to one. If there are still more leaves or polygons left to the process, the cycle repeats and vertex and index data continues to get collected for the next round of vertex and index buffers that will be created the next time CommitBuffers is called. The code to this function will be covered after we have finished discussing the BuildRenderData method.

The second new CBaseTree method called inside this loop is the WeldBuffers method. This function will be called whenever we fill up the temporary vertex array in an effort to compact the data and get rid of duplicated vertices. We have discussed welding many times throughout this course and understand that it will collapse multiple vertices that share identical properties into a single vertex. This potentially reduces the vertex count and will allow us to fit more vertices in the buffer. Of course, if the vertices are altered or compacted, the weld operation will also need to update the indices so that triangles that indexed vertices that have been removed are now mapped to the vertex they were collapsed onto. Once we find that we cannot add any more polygon vertices to the temporary vertex array without exceeding the maximum vertex count of the hardware, we will call the weld function to try to compact the vertex data in this array and make room for some more vertices. This function will need to be passed not only the temporary vertex array that will be compacted, but also the map of index buffers containing the indices currently collected for each subset that index into this vertex array. These index arrays will also have to be updated so that they correctly index into the compacted vertex array. The contents of this function will also be discussed shortly.

Continuing where we left off, we are now at the point where we have allocated an array to hold the vertices we are going to collect for the vertex buffer we are about to build. We have also allocated an array of indices to record the indices generated for each leaf bin (stored in the BinIndexData map).

We will now loop through each leaf and through each polygon in that leaf. If the current polygon being processed is either invalid or invisible, we skip it and continue to the next iteration of the polygon loop.

```
    // Now we actually build the renderable data for the TRAVERSAL ordered
    // leaf list.
    for ( LeafIterator = Leaves.begin();
          LeafIterator != Leaves.end();
          ++LeafIterator )
    {
        // Retrieve leaf
        pLeaf = (CBaseLeaf*)(*LeafIterator);
        if ( !pLeaf ) continue;

        // Loop through each of the polygons stored in this leaf
        nPolygonCount = pLeaf->GetPolygonCount();

        for ( i = 0; i < nPolygonCount; ++i )
        {
            CPolygon * pPoly = pLeaf->GetPolygon( i );

            // Skip if it's invalid
            if ( !pPoly || pPoly->m_nVertexCount < 3 ) continue;

            // Skip if it's not visible
            if ( !pPoly->m_bVisible ) continue;
```

At this point we have a pointer to the current polygon we are going to process. First we will assign a ULONG pointer to point at the index array in the BinIndexData map for the polygon's subset. We will also assign a pointer to point at the BinSizes value for this subset. This will be set to 0 if this is the first polygon with the current subset we have found so far in the loop.

```
            ULONG * pIndexData = BinIndexData[ pPoly->m_nAttribID ];
            ULONG * pBinSize   = &BinSizes[ pPoly->m_nAttribID ];
```

Just so we understand these variables, pIndexData points to the index buffer we allocated earlier for the leaf bin with the same attribute ID as the current polygon. This pointer will be used to store the indices of the polygon in the array. pBinSize will point to the value in the size map for this polygon's subset so that we can increment the size as we add indices to the index array. This means, with each polygon we add, pBinSize will always tell us how many indices we have currently collected in this subset's index array and therefore, at what location to place the new indices.

Our next task is to fetch the leaf bin that the polygon will be assigned to. We pass the polygon's attribute ID into the GetLeafBin method to retrieve a pointer to the leaf bin that is assigned to the polygon's subset and will ultimately receive its indices.

```
            // Retrieve the render data item for this leaf / attrib ID
            pLeafBin    = GetLeafBin( pPoly->m_nAttribID );
            pRenderData = pLeaf->GetRenderData( pPoly->m_nAttribID );
```

Notice in the above code how we also call the leaf's GetRenderData method to fetch the RenderData structure in the leaf for the current subset. Remember, there will ultimately be a RenderData structure stored in the leaf for every subset contained in that leaf as it is inside the RenderData structure that the leaf's triangle range (for that subset) is stored. Of course, at this point, the leaf may not have a RenderData structure allocated for it if this is the first polygon we have encountered in the current leaf that has this subset. Therefore, if a RenderData structure exists in the leaf for the passed subset, its pointer will be stored in pRenderData. If not, pRenderData will be assigned a value of NULL, which means we had better allocate one.

```
            // If we were unable to find a render data item yet built for this
            // leaf / attribute combination, add one.
            if ( !pRenderData )
            {
                pRenderData = pLeaf->AddRenderData( pPoly->m_nAttribID );
                if ( !pRenderData ) return false;

                // Store the leaf bin for later use
                pRenderData->pLeafBin = pLeafBin;

            } // End if no render data
```

As you can see in the above code, if the current leaf being processed does not yet have a RenderData structure allocated for it for the subset of the current polygon being processed, we allocate a new one in that leaf by using the CBaseLeaf::AddRenderData method and passing in the subset ID/leaf bin we would like it associated with. The function will return a pointer to a new RenderData structure which we

will store in the pRenderData local pointer. We will then use this pointer to assign the RenderData's pLeafBin member to point at the leaf bin that will contain its indices (which was fetched above)

At this point we have the polygon we are currently processing and we also have the RenderData structure associated with its subset for the current leaf we are processing. The m_nVBCount local variable describes the index of the vertex buffer we are currently collecting vertices and indices for. This will obviously be 0 at the start of the function since we are collecting vertices for vertex buffer[0] in the tree's vertex buffer array. Every time the vertex array is filled up and the CommitBuffers method is called to create the vertex and index buffers, this value will be incremented. For example, after the first set of buffers has been created, it will contain an index of 1 and we will know that any vertices we collect are going to be assigned to the tree's second vertex buffer. We also know that the next set of index buffers we create for the leaf bins will index into this second buffer.

Each RenderData structure in a leaf (remember, there will be one for each subset stored in that leaf), will contain an array of one or more Element structures. For example, if a leaf has a series of polygons belonging to the same set that ultimately ends up getting spread over two vertex buffers, the RenderData structure for that subset (in that leaf) will have two Element structures. Each Element contains the run of triangles for one of those vertex buffers.

So now that we know which RenderData item this polygon's triangles need to be added to, we have to get the Element structure within it for the current vertex buffer being compiled. In the next section of code we loop through each Element in the RenderData's element array to search for the one that has a vertex buffer index assigned to it that is equal to the vertex buffer we are currently compiling data for (m_nVBCount). When we find the correct element to add the polygon's triangles to, we break from the loop as shown below.

```
            // Search for element associated with the current vertex buffer
            for ( j = 0; j < pRenderData->ElementCount; ++j )
            {
                pElement = &pRenderData->pElements[j];
                if ( pElement->VBIndex == m_nVBCount ) break;

            } // Next Element
```

If, at the end of the above loop, the loop variable *j* is equal to the RenderData's current element count, it means the loop did not break early. This tells us that the RenderData structure does not currently have an Element for the vertex array currently being compiled. This also means the current polygon is the first we have found in this leaf that uses this subset since collecting data for the current vertex buffer. Thus we will have to add a new Element to the RenderData's element array.

The following code adds a new Element structure to the leaf's RenderData structure that matches the attribute ID of the polygon currently being processed. This is all handled in the CBaseLeaf::AddRenderDataElement function. It finds the RenderData structure in its array with an attribute that matches the polygon's and adds a new Element structure to the end of its element array. It then returns a pointer to this new element.

```
            // If we reached the end, then we found no element
```

```
            if ( j == pRenderData->ElementCount )
            {
                // Add a render data element for this vertex buffer
                pElement = pLeaf->AddRenderDataElement( pPoly->m_nAttribID );
                pElement->IndexStart     = *pBinSize;
                pElement->PrimitiveCount = 0;
                pElement->VBIndex        = m_nVBCount;

            } // End if no element for this VB
```

Notice in the above code how once we get back the new Element for the current vertex buffer, we set its primitive count to 0 as no triangles have been added to it yet. We also set its VBIndex member to the current value stored in m_nVBCount. This tells us the number of vertex buffers we have created so far and therefore, the index of the current vertex buffer we are collecting data for.

What requires a little explanation is why we set the IndexStart value of this element to the value currently contained in pBinSize. We saw just inside the polygon loop that this is used to point at the value in the row of the BinSizes map that records the current number of indices that have been collected so far for this attribute. This was set to zero for every subset at the beginning of the leaf loop and will be incremented every time we add indices to the corresponding index array. Therefore, the value pointed at by this pointer will always contain the number of indices we have added to that subset's temporary index array so far. Therefore, we know that the indices we are about to add for the current polygon we are processing will begin at that location in the buffer. To clarify, if the pBinSize pointer points to a value of 6 and the polygon we are currently processing belongs to subset 20, it means we have current added 6 indices to the temporary index array stored at BinIndexData[20]. When we add the current polygon's indices to this same array, and because we only enter the above section of code if we have just created a new element, this triangle must be the first triangle we have found in the leaf so far for the current vertex buffer being compiled. Thus, its first index will be the start index for the run of triangles that use this subset and may exist in this leaf.

At this point we have the polygon itself, the leaf bin in which it is contained, the RenderData structure within that leaf that pertains to the subset of the polygon, and the Element structure in that RenderData structure that references the current vertex buffer being compiled. We are now ready to start adding the vertices and indices in this polygon to their respective buffers.

Before we can add the vertices of any of the polygon's triangles to the vertex array (pVertices), we first make sure that there is room enough to do so. If the current number of vertices we have collected in the vertex array so far (nVertexCount) is greater than the maximum number of vertices we will be able to store in a single vertex buffer (nMaxVertexCount) minus 3 (we need at least three spare slots to add the next triangle), it is time to commit the vertex array and all the index arrays we have compiled for each leaf bin to Direct3D vertex and index buffers.

However, before we do, we will perform a weld on the vertex array just to make sure that we cannot compact the vertex data and make some more room in there. Therefore, in the next section of code, once we realize there is no more room left in the vertex array for another triangle, we call the CBaseTree::WeldBuffers method. As the job of this method is to remove duplicated vertices (and remap the indices accordingly), the function returns the new vertex count describing how many vertices are in

the passed vertex array after the weld has been performed. Since the function will also have the job of remapping all the index arrays that reference vertices that get collapsed in the weld process, we must also pass this function the map of index arrays (for each leaf bin).

```
            if ( nMaxVertexCount != nTotalVertices &&
                 nVertexCount > nMaxVertexCount - 3 )
            {
                // Weld the data built so far
                nVertexCount = WeldBuffers( nVertexCount,
                                            pVertices,
                                            BinIndexData,
                                            BinSizes );
```

We will look at the code to this function later, so for now just know that when it returns, the pVertices array may have had vertices removed. The return value (which we store in nVertexCount) describes the new number of vertices in this array. On function return, any index arrays stored in the BinIndexData map will have had some of their indices remapped to new vertex positions in the array if these indices referenced vertices that were collapsed during the weld operation.

After the weld has been performed it is entirely possible that we may have made more space in the vertex buffer and may now have enough space to add the triangles in this polygon. However, if the vertex count of the pVertices array is still too large and we cannot add another triangle, we must commit the buffers using the CBaseTree::CommitBuffers method as shown below.

```
                // If there is still not enough room commit buffers
                if ( nVertexCount > nMaxVertexCount - 3 )
                {
                    CommitBuffers( nVertexCount,
                                   pVertices,
                                   BinIndexData,
                                   BinSizes,
                                   b32BitIndices );


                   nVertexCount = 0;

                    // Add a new render data element and prepare it
                    pElement = pLeaf->AddRenderDataElement( pPoly->m_nAttribID );
                    pElement->IndexStart     = 0;
                    pElement->PrimitiveCount = 0;
                    pElement->VBIndex        = m_nVBCount;

                } // End if start a clean buffer.

            } // End if not enough room for even a single triangle.
```

The CommitBuffers method is passed the vertex array that we have compiled so far and the map containing all the index arrays we have compiled for this vertex array so far. It is also passed the size of the vertex array and the map describing the size of the index arrays contained in the BinIndexData map. Note that we also pass the Boolean that describes to this function whether or not it should create 32-bit or 16-bit index buffers.

The CommitBuffers method is not a large method, but it has a very important job. For starters, it creates a new vertex buffer and populates it with the data in the passed vertex array and then adds this vertex buffer to the tree's vertex buffer array. It will also loop through every row in the BinIndexData map and extract the index arrays compiled for each leaf bin (for the current vertex buffer). For each index array, it will create a new index buffer and populate it. Then it will add each index buffer to the correct leaf bin which should own it. After the index buffer for each leaf bin has been created, its corresponding size value in the BinSizes map will be reset to zero so that we can start collecting new indices for the next vertex buffer. This function will also increase the tree's m_nVBCount variable so that it reflects the new number of vertex buffers stored in the tree.

When the function returns, we wish to start the collection process all over again (the index array counters have already been reset back to zero in the BinSizes map by the CommitBuffers function). Therefore, we set the current vertex count back to zero so that the vertex array can be used again to collect vertices for the next vertex buffer. This means that any Elements we create from this point on (in the RenderData structures of each leaf) will describe triangle runs in the new vertex buffer we are about to build.

Finally, notice in the above code that once we have committed the buffers, we have essentially deduced that the polygon being processed will be added to a new Element in the leaf's RenderData structure (describing triangle runs in the new vertex buffer we are just about to compile). Therefore, we add a new Element to the RenderData structure that is linked to the new vertex buffer we are about to create. Remember, the CommitBuffers method would have incremented the value of m_nVBCount so that it now contains the index of the next vertex buffer we will create the next time CommitBuffers is called. We still have not added the polygon's indices to this Element yet, so we set the primitive count of this new structure to 0. Further, because we have just committed the buffers, we know this polygon's triangles will be positioned at the start of the index buffer that will be created for the new vertex buffer (for the leaf bin associated with the polygon attribute). Therefore, we can set the index start to zero as well.

As discussed many times before, we will add the polygon triangles to the index buffer by stepping around each vertex in the polygon. For each vertex we will add it to the vertex array and for every vertex beyond the second one, we will create indices for the first vertex in the polygon, the previous vertex in the edge, and the current vertex. These three indices will be added to the index array associated with this leaf bin and the pBinSize pointer will have the value it points at (the number of indices in the array) incremented each time an index is added.

In the first section you will see us setting up a loop to step through each vertex in the edge. Notice just above the loop that we store the number of vertices currently in the vertex array in the FirstVertex local variable. This is important because we will need to keep track of where this polygon's vertices are being added from so that we can construct the right indices that index them.

```
            // Store triangle pre-requisites
            ULONG FirstVertex    = nVertexCount;
            ULONG PreviousVertex = 0;

            // For each triangle in the set
            for ( j = 0; j < pPoly->m_nVertexCount; ++j )
```

```
            {
                // Add this vertex to the buffer
                pVertices[ nVertexCount++ ] = pPoly->m_pVertex[ j ];
```

If the current loop variable *j* is greater then or equal to 2, it means we have added at least three vertices from this polygon so far and we can start generating indices to describe its triangles. As Figure 15.4 illustrates, at vertex 3 we will add the indices for vertices 1, 2, and 3. At vertex 4 we will increment the previous vertex and the current vertex so that the next triangle indexes vertices 1, 3, and 4, and so on around the edges of the polygon.

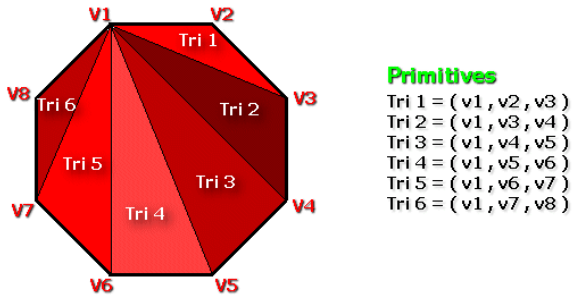



**Figure 15.4**

For every three indices we add to the index buffer we also know that we have just added a triangle (primitive) to the current element, so we increment the primitive count for that element. We then adjust the previous vertex by 1 so that it now points at the current vertex so that in the next iteration of the loop, we will have the next pair of vertices in the edge.

```
                // Enough vertices added to start building triangle data?
                if ( j >= 2 )
                {
                    // Add the index data
                    pIndexData[ (*pBinSize)++ ] = FirstVertex;
                    pIndexData[ (*pBinSize)++ ] = PreviousVertex;
                    pIndexData[ (*pBinSize)++ ] = nVertexCount - 1;

                    // Update leaf element, we've added a primitive
                    pElement->PrimitiveCount++;

                } // End if add triangle data

                // Update previous vertex
                PreviousVertex = nVertexCount - 1;
```

Because we are adding the polygons to our index buffers one triangle at a time, we may find that we might be only part of the way through adding its triangles when we fill up the vertex buffer. Looking at Figure 15.4 for example, we can see how the first vertex might be v1, the previous vertex might be v4 and the current vertex might be v5, which collectively describe triangle 3. However, after adding triangle 3, we might find that the vertex buffer is full and we have to switch buffers. If this is the case, we will first call the weld function to try compacting the vertex array, hopefully making enough room to add all the vertices in this polygon to the vertex array. Notice this time that we pass in two additional parameters to the WeldBuffers function.

```
                if ( nMaxVertexCount != nTotalVertices &&
                     nVertexCount == nMaxVertexCount )
                {
                    // Weld the data built so far
                    nVertexCount = WeldBuffers( nVertexCount,
                                                pVertices,
```

```
                                                BinIndexData,
                                                BinSizes,
                                                &FirstVertex,
                                                &PreviousVertex );
```

Because we are in the middle of stepping around the vertices of a triangle when we perform the weld, we will need to know where the position of the first vertex and the previous vertex in the edge got moved to during the weld. This tells us which vertices to use for the next triangle. For example, imagine we have just added triangle 3 in Figure 15.4 where the first vertex (v1) was at position 20 in the vertex buffer and the previous vertex (which was v4 when the triangle was added and was then increased to v5) was set to 24. We know that if the weld had not been performed, the next triangle added would consist of vertices 20, 24, and 25. However, after the weld, vertices 20 and 24 might get repositioned in the buffer to locations 10 and 14, in which case we will need to update the values stored in FirstVertex and PreviousVertex to these values so that the next triangle added will use vertices 10 and 14 and the next vertex in the edge. That is what the last two values to the weld function are for. The weld function will return the new positions of these vertices using these output parameters. That is, FirstVertex and PreviousVertex tell the weld function the two vertices we wish to track the changes for. After the weld is complete, these values will be updated with the new vertex positions.

Of course, if the weld function did not successfully make enough room in the vertex array for the next vertex in the polygon, we will need to commit the buffers and start collecting vertices and indices for the next vertex buffer.

```
                    // If there is still not enough room, commit the changes
                    if ( nVertexCount == nMaxVertexCount )
                    {
                        // Commit the changes
                        CommitBuffers( nVertexCount,
                                       pVertices,
                                       BinIndexData,
                                       BinSizes,
                                       b32BitIndices );

                        nVertexCount = 0;
```

What we have to remember is that if a polygon spans multiple buffers, we will need to also add the first vertex and previous vertex to the new buffer as well. For example, imagine in Figure 15.4 we have added triangle 1 and then we start a new buffer. The next vertex in the polygon to be processed would be v4 and the triangle that should be added to the new buffer would be <v1, v3, v4>. However, v1 and v3 exist in the previous vertex buffer so they cannot possibly be indexed by the new index buffers we are about to create. Therefore, we must make sure that these vertices exist in the new buffer as well.

In the next section of code we can see that if we do change buffers mid-polygon, we add a new element for that vertex buffer in the RenderData structure associated with the polygon's subset. We also assign its VBIndex member so that it describes the next vertex buffer that will be created the next time CommitBuffers is called. As we have not added any primitives yet, we will set its index start and primitive count to zero.

```
                        // If the polygon spans multiple VB's,
                        // then we need a new element
                        if ( (signed)j < pPoly->m_nVertexCount - 1 )
                        {
                            // Add a new render data element and prepare it
                          pElement=pLeaf->AddRenderDataElement(pPoly->m_nAttribID);
                          pElement->IndexStart     = 0;
                          pElement->PrimitiveCount = 0;
                          pElement->VBIndex        = m_nVBCount;
```

We will now add the first vertex in the polygon and the current vertex (which will become the previous vertex in the next iteration) to the first element of the new vertex array. We will then set this vertex array's count to 2 since we have only collected two vertices for this vertex array so far. We then assign the FirstVertex and PreviousVertex variables to contain the indices of the first and second vertices in the new vertex array so that in the next loop, the vertices we just duplicated in the new buffer are used to construct the next triangle. The remainder of the entire process is shown below.

```
                            // Since this polygon data is now split over multiple
                            // vertex buffers, we're going to need to re-add the
                            // first and previous vertices to the
                            // new buffer as well. This is so that we can form a
                            // valid triangle next time assuming there is any
                            // triangle data left to be processed for this polygon,

                            pVertices[0]   = pPoly->m_pVertex[ 0 ];
                            pVertices[1]   = pPoly->m_pVertex[ j ];
                            nVertexCount   = 2;
                            FirstVertex    = 0;
                            PreviousVertex = 1;

                        } // End if more triangles to come

                    } // End if not enough room.

                } // End if buffer full

            } // Next Triangle

        } // Next Polygon

    } // Next Leaf
```

At this point in the function we have processed every leaf and every polygon in the tree. We will have constructed vertex buffers and added them to the tree and we will have added (potentially) multiple index buffers to each leaf bin. Every leaf will contain a RenderData structure for each subset it contains, which in turn will contain an array of elements for every vertex buffer that subset (within that leaf) is contained within.

Of course, as the buffers are only committed once they are full, unless the very last vertex we added caused the buffers to be committed, there will still be vertices in the pVertices array and index arrays in the BinIndexData map that have not yet been committed. Therefore, as our final step, we weld any

vertices we have not yet committed before committing them too, creating one last vertex buffer and one last set of index buffers (one per leaf bin which has indices in the index arrays for this vertex buffer).

```
    // Build any remaining data
    if ( nVertexCount > 2 )
    {
        // Weld any remaining data and commit it
        nVertexCount = WeldBuffers( nVertexCount,
                                    pVertices,
                                    BinIndexData,
                                    BinSizes );

        CommitBuffers( nVertexCount,
                       pVertices,
                       BinIndexData,
                       BinSizes,
                       b32BitIndices );

    } // End if any data left to commit
```

With our render data constructed we must release all temporary memory that was not allocated on the stack. So we will iterate through each row in the BinSize map (one for each leaf bin) and extract its attribute ID and index array size. If this size if greater then zero, it means there is a corresponding index array in the BinIndexData map that needs to be deleted from memory. Thus we fetch the index array from the BinIndexData map with the matching attribute ID and call delete on its pointer.

```
    // Release temporary index buffer arrays
    BinSizeIterator = BinSizes.begin();
    for ( ; BinSizeIterator != BinSizes.end(); ++BinSizeIterator )
    {
        ULONG AttribID = BinSizeIterator->first;
        ULONG Size     = BinSizeIterator->second;
        if ( Size > 0 && BinIndexData[ AttribID ] )
            delete []BinIndexData[ AttribID ];

    } // Next Bin
```

We now delete the vertex array that was used to collect the vertices for each vertex buffer before returning a successful build indicator.

```
    // Release temp vertex array.
    if ( pVertices ) delete []pVertices;

    // Success!
    return true;
}
```

This was some very complicated code to try to absorb in one read, so fully expect to have to make a few passes over the source code to this function (and the explanations included here) to fully understand the relationships between the various components.

One of the major processes in the above function that we have yet to cover is the call to the CommitBuffers method. The code to this method will be discussed next.

## CommitBuffers - CBaseTree

The function is passed an array of vertices which must be used to create a new vertex buffer for the render system. It will also be passed two STL maps. The first map's rows will contain an index array for each leaf bin. The number of rows in this map will be equal to the number of leaf bins used by the tree. This function will create an index buffer for each of these index arrays and store each index buffer in the relevant leaf bin. The second map that is passed (parameter four) describes the number of indices contained in each of the index arrays. The final parameter is a Boolean which tells the function whether the index buffer it creates for each leaf bin should use 16 or 32-bit indices.

The first thing the function does is grow the tree's vertex buffer array making room for an extra vertex buffer pointer at the end of the array. If there are current vertex buffer pointers in this array, this involves the allocation of a new array large enough to hold all the old buffer pointers plus the space at the end for the new buffer. The buffer pointers are then copied over from the old array into the new one and the old array is released. The tree's pointer to its vertex buffer array is then updated to point at this new array and the vertex buffer count is then increased.

```
bool CBaseTree::CommitBuffers( ULONG VertexCount,
                               CVertex * pVertices,
                               map<ULONG, ULONG*> & BinIndexData,
                               map<ULONG, ULONG> & BinSizes,
                               bool b32BitIndices )
{
    ULONG                      ulUsage = D3DUSAGE_WRITEONLY;
    LPDIRECT3DVERTEXBUFFER9 * ppVBBuffer, pVB;
    CVertex                 * pDestVertices;
    UCHAR                   * pDestIndices;
    ULONG                   * pIndices;
    HRESULT                    hRet;
    ULONG                      i;

    // First allocate space for a new vertex buffer
    ppVBBuffer = new LPDIRECT3DVERTEXBUFFER9[ m_nVBCount + 1 ];
    if ( !ppVBBuffer ) return false;

    // Any old data?
    if ( m_nVBCount > 0 )
    {
        // Copy over the old data, and release the old array
        memcpy( ppVBBuffer,
                m_ppVertexBuffer,
                m_nVBCount * sizeof(LPDIRECT3DVERTEXBUFFER9) );

        delete []m_ppVertexBuffer;

    } // End if any old data
```

```
    // Set the new entry to NULL
    ppVBBuffer[ m_nVBCount ] = NULL;

    // Store the new buffer pointer
    m_ppVertexBuffer = ppVBBuffer;
    m_nVBCount++;
```

Notice in the above code that we assign the ulUsage variable the value of D3DUSAGE_WRITEONLY. This variable will be used for the vertex buffer creation flags. We will inform Direct3D to create the vertex buffer in write only mode so that memory placement of the vertex buffer is chosen based on the best performance. If the current device is a software device (which will be stored in the tree's m_bHardwareTnL member passed into its constructor), the D3DUSAGE_SOFTWAREPROCESSING flag will need to be combined with ulUsage so that the vertex buffer is created in system memory (for efficient CPU transformation and lighting).

Next we allocate a new vertex buffer large enough to hold the number of vertices contained in the passed array. Allocation will be in the managed resource pool. We then store the newly create vertex buffer's pointer in the tree's vertex buffer array, as shown below.

```
    // Should we use software vertex processing ?
    if ( !m_bHardwareTnL ) ulUsage |= D3DUSAGE_SOFTWAREPROCESSING;

    // Create the actual vertex buffer ready to store the data
    hRet = m_pD3DDevice->CreateVertexBuffer( VertexCount * sizeof(CVertex),
                                             ulUsage,
                                             VERTEX_FVF,
                                             D3DPOOL_MANAGED,
                                             &pVB,
                                             NULL );
    if ( FAILED(hRet) ) return false;

    // Store the vertex buffer in the array
    // (so that it's released if anything goes wrong)
    m_ppVertexBuffer[ m_nVBCount - 1 ] = pVB;
```

With the new vertex buffer allocated and added to the tree's vertex buffer array, we then lock the vertex buffer and copy in all the vertices in the input vertex array before unlocking it.

```
    // Lock the vertex buffer ready to copy
    hRet = pVB->Lock( 0, 0, (void**)&pDestVertices, 0 );
    if ( FAILED(hRet) ) return false;

    // Copy over the data that we were passed
    memcpy( pDestVertices, pVertices, VertexCount * sizeof(CVertex) );

    // Unlock the buffer, we're done with VB creation
    pVB->Unlock();
```

The vertex buffer has now been dealt with. Next we have to create the index buffers for each leaf bin. First we set up a loop to iterate though each row in the BinIndexData map (one row for each leaf bin)

and extract the attribute ID for that row. This tells us what leaf bin the index array should be assigned to. We also extract the size of the index array (stored in the BinSizes map with the same key) and fetch a pointer to the leaf bin assigned to that attribute using the CBaseTree::GetLeafBin function.

```
    // Now that we've built the VB, we need to propogate the index data
    map<ULONG, ULONG*>::iterator DataIterator = BinIndexData.begin();
    for ( ; DataIterator != BinIndexData.end(); ++DataIterator )
    {
        ULONG AttribID = DataIterator->first;
        ULONG Size     = BinSizes[ AttribID ];

        // Retrieve the leaf bin
        CLeafBin * pLeafBin = GetLeafBin( AttribID );
        if ( !pLeafBin ) { BinSizes[AttribID] = 0; continue; }
```

Every index buffer added to a leaf bin is contained inside its own CLeafBinData structure inside the leaf bin's internal array of these structures. Thus, we allocate a new CLeafBinData structure.

```
        // Allocate a new LeafBinData structure
        CLeafBinData * pData = new CLeafBinData;
        if ( !pData ) return false;
```

We now instruct the leaf bin to add this CLeafBinData pointer to its internal array of CLeafBinData pointers.

```
        // Add to leaf bin, so that the data is released if something goes wrong
        if ( !pLeafBin->AddLeafBinData( pData ) ) { delete pData; return false; }
```

It is time to fill out the information for the CLeafBinData structure, including whether or not it uses 32-bit indices, the number of triangles that will be in the index buffer assigned to this CLeafBinData structure, and the index of the vertex buffer in the tree's vertex buffer array that this index buffer will index into. We also store the number of vertices in that vertex buffer in the structure.

```
        // Fill out the data items for this leaf bin.
        pData->m_b32BitIndices = b32BitIndices;
        pData->m_nFaceCount    = Size / 3;
        pData->m_nVBIndex      = m_nVBCount - 1;
        pData->m_nVertexCount  = VertexCount;
```

If the index array inside the BinIndexData map for this leaf bin has no indices, we will just skip generating an index buffer for this leaf bin (for this vertex buffer) and continue on to process the next leaf bin.

```
        // If there is no data here,
        // we'll just continue (but we must have allocated the leaf bin data)
        if ( Size == 0 ) continue;
```

If we get this far, it indicates that there are indices for the current leaf bin, so we will need to create an index buffer.

```
        // Create the index buffer ready to store the data
        UCHAR     IndexStride = ( b32BitIndices ) ? 4 : 2;
        D3DFORMAT Format      = (b32BitIndices) ? D3DFMT_INDEX32 : D3DFMT_INDEX16;

        hRet = m_pD3DDevice->CreateIndexBuffer( Size * IndexStride,
                                                ulUsage,
                                                Format,
                                                D3DPOOL_MANAGED,
                                                &pData->m_pIndexBuffer,
                                                NULL );
        if ( FAILED(hRet) ) return false;
```

With the index buffer for this leaf bin/vertex buffer combination allocated, we will now lock it and copy over all the index data from the index array for this leaf bin contained in the BinIndexData map.

```
        // Lock the index buffer ready to copy
        hRet = pData->m_pIndexBuffer->Lock( 0, 0, (void**)&pDestIndices, 0 );
        if ( FAILED(hRet) ) return false;

        // Copy over the buffer data
        if ( b32BitIndices )
        {
            // We can do a straight copy if it's a 32 bit index buffer
            pIndices = DataIterator->second;
            memcpy( pDestIndices, pIndices, Size * IndexStride );

        } // End if 32bit
        else
        {
            pIndices = DataIterator->second;
            for ( i = 0; i < Size; ++i )
            {
                // Cast everything down to 16 bit
                ((USHORT*)pDestIndices)[i] = (USHORT)pIndices[i];

            } // Next Index

        } // End if 16bit

        // Unlock the buffer, we're done with IB creation

        pData->m_pIndexBuffer->Unlock();
```

Notice in the above code that when we create the index buffer, we store the returned pointer in the leaf bin's CLeafBinData structure.

Next we assign the address of the vertex buffer we have just created to the CLeafBinData's vertex buffer pointer so that this structure now contains pointers to both the vertex buffer and the index buffer that should be bound to the device in order to draw its render batches. Since we are making a copy of the vertex buffer pointer, we also increase its reference count.

```
        // Store a reference to the VB in the data area to make it easy to process
        pData->m_pVertexBuffer = pVB;
```

```
        pVB->AddRef();
```

Finally, having committed the index array for the current leaf bin to an index buffer, we set the size of this array back to zero so that when the function returns we can start collecting indices from the beginning of this array for the next vertex buffer. That ends the loop that is performed for each leaf bin.

```
        // Reset bin size to 0, we've commited the data
        BinSizes[ AttribID ] = 0;

    } // Next set of index data

    // Success!
    return true;
}
```

When the loop ends, every leaf bin that had indices collected for it for the current vertex buffer will have had an index buffer created for it and assigned to it. With the job done, the function then returns a success flag.

## WeldBuffers - CBaseTree

We saw this method being called from the BuildRenderData method discussed previously. It is used to perform a weld on the passed vertex array to (hopefully) compact the data and make some more room in the buffer for additional vertices. We call this function whenever the buffer is found to be full so that we are able to squeeze as many triangles into a single vertex buffer as possible. Fewer vertex buffers results in reduced vertex T&L and a smaller number of index and vertex buffer changes during rendering.

The function accepts six parameters. The first two parameters describe the size and contents of the passed vertex array that is going to be welded by this function. The second two parameters are maps that contain the index arrays collected for each leaf bin which indexed into the vertex array and the size of each leaf bin's index array. We must pass the index arrays collected for each leaf bin because they index into the passed vertex array. If the vertex array is going to be compacted and have duplicated vertices collapsed into a single vertex, the index arrays must also be updated so that any indices in any leaf bins' array that reference these deleted vertices are properly updated to reference the new vertices that the original vertices were collapsed onto. The final two parameters are optional. They allow us to pass the address of two vertex indices that will have their values tracked and remapped by the weld and returned to the caller. For example, if we passed in variables that held the values 12 and 24 respectively, this tells the weld function that we would like to know where the $12^{th}$ and $24^{th}$ vertex were repositioned in the vertex array after the weld. If these two vertices were relocated to slots 5 and 6 in the vertex buffer after the weld, on function return, the variable passed as the fifth parameter would contain 5 and the variable passed as the sixth parameter would contain the value 6. We saw why we needed to track the remapping of certain vertices during the BuildRenderData function. If a weld was performed partway through adding the triangles of a polygon to the vertex buffer, we needed to track where the two vertices used in the last triangle (that will also be used in the next triangle) will be positioned after the weld. This allows us to reference these vertices as we add the rest of the triangles in the polygon using their new vertex

buffer positions. All other times that a weld is performed in the BuildRenderData method, NULL is passed for the final two parameters since we only need to track the vertex positions when the weld happens mid-way through adding a single polygon.

Performing a weld seems like it would involve a good amount of code as you can imagine the various conditional tests that would be needed and we would certainly want a fast way of determining which vertices are candidates for collapse. While it would not be very difficult to code from scratch, fortunately D3DX includes a welding function that works with D3DX meshes and it is quite fast. All we will have to do is temporarily build a D3DX mesh using the vertex and index data passed in, and perform a weld on the mesh and extract the vertex and index data back out again into their original arrays which can then be returned to the caller. The mesh can then be released since it was only used temporarily so that we would get access to the D3DX weld functionality. Because the vertex array will be welded and all the index arrays (one for each leaf bin) describe triangles in that vertex array, we can think of the vertex array and all the index arrays combined as describing a single mesh. Therefore, what we will do is create a mesh into which we will copy all the vertex data in the vertex array into its vertex buffer and all the indices from all the index arrays (collected for each leaf bin in the BinIndexData map) into its index buffer. This means we will need to know how large and allocation we will for the temporary mesh's index buffer so that we can inform the mesh creation function of how many triangles it will need to store.

The first thing we do in this function is loop through all the rows in the BinSize map and sum up the sizes of each index array (one per leaf bin). We divide the total size by three (# indices per triangle) so that at the end of the loop we know the total number of triangles we will need to store in the mesh. Remember, this mesh will contain all of the triangles contained in all of the index arrays we have collected for each leaf bin/subset that reference the passed vertex buffer.

```
ULONG CBaseTree::WeldBuffers( ULONG VertexCount,
                              CVertex * pVertices,
                              map<ULONG,ULONG*> & BinIndexData,
                              map<ULONG,ULONG> & BinSizes,
                              ULONG * pFirstVertex /* = NULL */,
                              ULONG * pPreviousVertex /* = NULL */ )
{
    ULONG       i;
    HRESULT     hRet;
    LPD3DXMESH  pMesh;
    ULONG       TotalTriangles = 0;
    ULONG       nFirstIndex = 0xFFFFFFFF, nPreviousIndex = 0xFFFFFFFF, Counter = 0;

    // Total the full amount of triangles in the bin data so far
    map<ULONG,ULONG>::iterator SizeIterator = BinSizes.begin();
    for ( ; SizeIterator != BinSizes.end(); ++SizeIterator )
    {
        TotalTriangles += (SizeIterator->second / 3);

    } // Next Index Buffer
```

At this point the TotalTriangles local variable will contain all the triangles described by every index array. We now create a D3DXMesh that is large enough to store the correct number of vertices and triangles.

```
    hRet = D3DXCreateMeshFVF( TotalTriangles,
                              VertexCount,
                              D3DXMESH_SYSTEMMEM |
                              D3DXMESH_SOFTWAREPROCESSING | D3DXMESH_32BIT,
                              VERTEX_FVF,
                              m_pD3DDevice,
                              &pMesh );

    if ( FAILED( hRet ) ) return false;
```

Notice that since we are using this mesh for CPU bound operations (we do not intend to render it) we pass in the D3DXMESH_SYSTEMMEM and D3DXMESH_SOFTWAREPROCESSING flags so that we have a mesh created in system memory.

Next, we lock its vertex buffer, its index buffer, and its attribute buffer because we will need to fill the mesh with the data from our vertex and index arrays.

```
    ULONG   *pDestIndices, *pAttributes;
    CVertex *pDestVertices;

    // Lock the vertex, index and attribute buffers ready for population
    if ( FAILED( pMesh->LockIndexBuffer( 0, (void**)&pDestIndices ) ))
         { pMesh->Release(); return VertexCount; }

    if ( FAILED( pMesh->LockVertexBuffer( 0, (void**)&pDestVertices ) ))
         { pMesh->Release(); return VertexCount; };

    if ( FAILED( pMesh->LockAttributeBuffer( 0, &pAttributes ) ))
         { pMesh->Release(); return VertexCount; }
```

We copy the vertices in the passed array into the mesh's vertex buffer and then set every entry in the mesh's attribute buffer to zero.

```
    // Copy over the vertices
    memcpy( pDestVertices, pVertices, VertexCount * sizeof(CVertex) );

    // Zero out the attribute buffer
    memset( pAttributes, 0, TotalTriangles * sizeof(ULONG) );
```

Why do we fill the attribute buffer with zeros? We do this because the D3DX weld function may perform operations on the indices that cause them to be moved around in the index buffer. For example, triangles may be shuffled around so that they are grouped by subset. The problem is, although we wish the vertex data to be compacted and the indices to have their values updated to correctly use these vertices, we must be absolutely sure that the weld function never changes the location of the triangles within the mesh's index buffer. We need to know that the triangles stay exactly where they are so that we can extract them out again into the index arrays for each leaf bin. Imagine for example that we added

10 triangles from leaf bin 1 followed by 10 triangles from leaf bin 2 to the mesh's index buffer. Provided the triangles never have their positions within the index buffer changed, we know that after the weld we can extract the first 10 triangles back out into leaf bin 1's array and the second 10 back out into leaf bin 2's index array. However, if these triangles were repositioned, we would lose all knowledge of which triangles in the welded index buffer need to be copied back out into their original index array. By placing all zeros in the mesh's attribute buffer, we trick the mesh into thinking that all the triangles belong to a single subset (subset zero) and therefore, the optimization that moves triangles for better subset batching will not be performed.

Our next task is to loop through every row in the BinIndexData map and extract the index pointer for each subset/leaf bin. We then copy the indices stored in each index array into the mesh's index buffer. The next section of code starts the beginning of this loop.

```
    // Now we need to update the indices
    map<ULONG,ULONG*>::iterator BinIterator = BinIndexData.begin();
    for ( ; BinIterator != BinIndexData.end(); ++BinIterator )
    {
        ULONG AttribID = BinIterator->first;
        ULONG Size     = BinSizes[ AttribID ];

        // Skip if there is no data
        if ( Size == 0 ) continue;

        // Copy over the indices
        ULONG * pIndices = BinIterator->second;
        memcpy( pDestIndices, pIndices, Size * sizeof(ULONG) );
```

At this point in the current iteration of the loop we have copied over all the indices of the current index array we are processing into the mesh's index buffer (pDestIndices).

The next section of the loop code may look a little strange, so let us quickly explain why it is needed. As the final two input parameters, the caller can pass the address of two unsigned longs that describe vertex indices into the original vertex array passed. We know that the weld function will automatically update all the indices in the index buffer so that the triangles correctly index the updated vertices in the post-weld vertex buffer (even if many of the original vertices were deleted/collapsed). However, if we call the weld function partway though adding the triangles of a single polygon to the vertex array (which we might do in the BuildRenderData method), we will need to track the position changes made to two of the polygons vertices. This way, when the weld function returns, we can continue to add another triangle using the two vertices used in the previous triangle that was added before the weld function was called. We explained where and why we need this functionality when we covered the BuildRenderData method earlier, so refer back to that discussion if you are fuzzy on the details.

If either pFirstVertex or pPreviousVertex are not NULL, it means they point to variables assigned by the caller that contain the indices of two vertices. It also means that the caller would like to know where these two vertices end up in the vertex array after the weld. Unfortunately, the weld function does not give us this information, so we will have to do a little work beforehand. Essentially, as we add each index array to the index buffer, we will loop through those indices searching for an index that references the vertex we are interested in tracking the position changes for. For example, if *pFirstVertex=10 it

means the caller would like to know the new index for vertex 10 if it got moved in the vertex array during the weld. In order to do this, we test each index array as we add it and search for the first index we find that references this vertex. We then record the location of this index. After the weld, the vertices may have changed, but the index we recorded will still be in the same location in the index buffer, so we can read back the value it now contains. This will tell us the vertex that index has been updated to use and the new position of the original vertex 10 in the new vertex buffer.

Let us just step through an example so that the logic is clear. Let us say that the caller would like to know the post-weld positions of vertex 10 and vertex 12. They would pass in as the pFirstVertex and pPreviousVertex parameters pointers to two variables that contain the values 10 and 12, respectively. In this next section, we will search for the first index in any index buffer that currently indexes vertices 10 and 12 in the vertex array and we will record the locations of these indices in the index buffer. Let us imagine that we found that the first index that references vertex 10 is in leaf bin 2's index array at location 100 in the mesh's main index buffer. We will record this index location. Let us also assume that we find the first reference to vertex 12 in leaf bin 5 and record the location where it has been added to the mesh's index buffer as well. Assume that it has been placed at index 500 in the mesh's index buffer. So we have the following.

*pFirstVertex = 10
*pPreviousVertex = 12

nFirstIndex = 100
nPreviousIndex = 500

Just to be clear, we have determined that there is an index that references vertex 10 stored at location 100 in the mesh's index array and an index that references vertex 12 at location 500 in the mesh's index array. Once we have recorded these values, we no longer need to search for it anymore; we are just interested in finding the first occurrence of these two vertices in the index buffer. After the weld is performed, vertices 10 and 12 may be in completely different locations. However, the triangles (and thus their indices) in the mesh index buffer will not have changed position at all, but they will have had their values updated by the weld function so that they correctly reference the updated vertex buffer. Therefore, all we have to do is read back the values now stored in the index buffer at locations 100 and 500 and we will have the new locations of vertex 10 and 12 in the vertex buffer. We can then return this information to the caller.

Let us look at the code in this loop that searches and records the index locations prior to the weld being performed. Notice that we only perform the search if we have not already found the first index that references each vertex.

```
        if ( (pFirstVertex && nFirstIndex == 0xFFFFFFFF) ||
             (pPreviousVertex && nPreviousIndex == 0xFFFFFFFF) )
        {
            // Loop through the indices
            for ( i = 0; i < Size; ++i )
            {
                if ( (pFirstVertex &&
                      nFirstIndex == 0xFFFFFFFF)
                      && pIndices[i] == *pFirstVertex ) nFirstIndex = Counter + i;
```

```
                if ( (pPreviousVertex &&
                      nPreviousIndex == 0xFFFFFFFF)
                      && pIndices[i] == *pPreviousVertex) nPreviousIndex=Counter+i;

            } // Next Index

        } // End if search for remap
```

At the start of the code, nFirstIndex and nPreviousVertex are set to 0xFFFFFFFF which means we have not yet found the location of an index that references the vertex. Notice we only step into this code block if we have not yet found both nFirstVertex and nSecondVertex and if the application did not pass NULL as the pFirstVertex and pPreviousVertex parameters. Inside the code block we loop through every index in the index array we are about to add and test to see if its value is equal to pFirstVertex. If it is, then we have found an index that references the first vertex we are interested in tracking, so we record the location of that index in the nFirstIndex variable. As this variable will no longer be set to 0xFFFFFFFF, the search for this index will not be performed again when adding any other vertex arrays. We do exactly the same thing for the pPreviousVertex and record the location of the first index we find that references it in nPreviousVertex. Notice above that the Counter variable starts at zero and is incremented at the bottom of the loop by the size of the index array we just added to the mesh's index buffer. Therefore, at the beginning of each loop iteration, it contains the location of first index in the index buffer of the index array we just added. Thus, Counter + i describes exactly where the index we have just found will be located in the mesh's index buffer (pre- and post-weld).

Finally, at this point in the loop we have added the current index array to the index buffer and have performed a search through those indices for the tracked vertices, so we increment the index buffer pointer past the new indices we just added so that it points at the location where the next index array that we process in the next iteration of the loop should be added. We also increment the Counter variable by the size of the index array so that it always describes (at the beginning of each loop iteration) the location in the index buffer where the index array that has just been added begins.

```
        // Move the destination index array on for the next batch
        pDestIndices += Size;
        Counter      += Size;

    } // Next Index Buffer
```

Once this has been done, we unlock all the buffers in the mesh since it now contains all the vertex and index data needed to perform the weld.

```
    // Unlock the buffers
    pMesh->UnlockIndexBuffer();
    pMesh->UnlockVertexBuffer();
    pMesh->UnlockAttributeBuffer();
```

You will recall from Chapter 8 that the D3DXWeld function must be passed a D3DXWELDEPSILONS structure that allows us to control how fuzzy the comparison is between two vertices. This allows us some control over what are considered duplicate vertices. We can assign the various vertex components

different epsilons so that two vertices that are almost the identical, but have positions that vary by as little as 0.01 (for example) are considered to be the same and are collapsed into a single vertex. Another example would be two vertices that share the exact same location and have normals that are similar enough to warrant welding them into a single vertex.

The D3DXWELDEPSILONS structure is shown below as a reminder. The larger the epsilon we assign to a vertex component, the more fuzzy the compare will be when determining whether two vertices are the same for that component.

```
typedef struct _D3DXWELDEPSILONS
 {
    FLOAT Position;
    FLOAT BlendWeights;
    FLOAT Normal;
    FLOAT PSize;
    FLOAT Specular;
    FLOAT Diffuse;
    FLOAT Texcoord[8];
    FLOAT Tangent;
    FLOAT Binormal;
    FLOAT Tess Factor;
} D3DXWELDEPSILONS;
```

We will set all the members of this structure to a standard epsilon of 0.001. Rather than assign each value of the structure to 0.001 individually, we know that it really is just a block of memory representing 10 floats, so we will just instantiate the structure, access it via a float pointer, and loop through the 10 floats assigning each location a value of 0.001 (1e-3).

```
    // Set all epsilons to 0.001;
    D3DXWELDEPSILONS WeldEpsilons;
    float * pFloats = (float*)&WeldEpsilons;
    for ( i = 0; i < sizeof(D3DXWELDEPSILONS) / sizeof(float); i++ )
         *pFloats++ = 1e-3f;
```

Now we pass this structure, along with the mesh pointer itself, into the D3DXWeldVertices method to perform the weld.

```
    // Weld the vertex data
    hRet = D3DXWeldVertices( pMesh,
                             D3DXWELDEPSILONS_WELDPARTIALMATCHES |
                             D3DXWELDEPSILONS_DONOTSPLIT,
                             &WeldEpsilons, NULL, NULL, NULL, NULL );

    if ( FAILED(hRet) ) { pMesh->Release(); return VertexCount; }
```

The D3DXWELDEPSILONS_WELDPARTIALMATCHES flag instructs the function that we would like duplicated vertices removed. The D3DXWELDEPSILONS_DONOTSPLIT flag instructs the function to never split (i.e., duplicate) vertices that are in separate attribute groups. This is passed just to be safe, since splits should never happen because we have zeroed out the attribute buffer. As far as the mesh is concerned, it only has a single subset.

With the weld performed, we now need to exact the mesh data back out into the vertex and index arrays that were passed into the function so that the modified data is returned to the caller. The first step is to lock the mesh vertex and index buffers and update the VertexCount parameter so that it now reflects the number of vertices in the welded mesh.

```
    // Lock the vertex and index buffers ready for extraction
    if ( FAILED( pMesh->LockIndexBuffer( 0, (void**)&pDestIndices ) ))
         { pMesh->Release(); return VertexCount; }

    if ( FAILED( pMesh->LockVertexBuffer( 0, (void**)&pDestVertices ) ))
         { pMesh->Release(); return VertexCount; };

    // Store updated vertex count
    VertexCount = pMesh->GetNumVertices();
```

In the next section we copy the vertices from the mesh's vertex buffer into the passed pVertices array. This overwrites the old vertex data contained there. We then access the indices in the index buffer at locations nFirstIndex and nPreviousIndex to retrieve the new locations of vertices described by the pFirstVertex and pPreviousVertex parameters. The variables addressed by these two pointers have their values updated to contain the new post-weld position of these vertices, which will be accessible to the caller on function return.

```
    // Copy over the vertices
    memcpy( pVertices, pDestVertices, VertexCount * sizeof(CVertex) );

    // Update remap information if it was requested
    if ( pFirstVertex    ) *pFirstVertex    = pDestIndices[ nFirstIndex ];
    if ( pPreviousVertex ) *pPreviousVertex = pDestIndices[ nPreviousIndex ];
```

Finally, all that is left to do is iterate through each index array in the BinIndexData map and overwrite their indices with the modified mesh indices.

```
    // Now we need to copy back the index buffers
    BinIterator = BinIndexData.begin();
    for ( ; BinIterator != BinIndexData.end(); ++BinIterator )
    {
        ULONG AttribID = BinIterator->first;
        ULONG Size     = BinSizes[ AttribID ];

        // Skip if empty
        if ( Size == 0 ) continue;

        // Copy over the indices
        memcpy( BinIterator->second, pDestIndices, Size * sizeof(ULONG) );

        // Move the destination index array on for the next batch
        pDestIndices += Size;

    } // Next Index Buffer

    // Unlock the buffers and release the mesh
    pMesh->UnlockIndexBuffer();
```

```
    pMesh->UnlockVertexBuffer();
    pMesh->Release();

    // Return new vertex count.
    return VertexCount;
}
```

At the end of this process, we unlock the mesh's buffers and release the mesh before returning the new modified vertex count of the passed vertex array back to the caller.

We have now covered all the code involved in preparing the data for rendering and all of the processes invoked by the CBaseTree::BuildRenderData function. When the BuildRenderData method returns, all the tree's vertex buffers and leaf bins will have been constructed and are ready for use. In the next section we will examine the code that manages the visibility pass on the tree and the code for rendering that visible data.

## 15.4.3 Processing Tree Visibility / Rendering the Static Data

Before the application instructs the spatial tree to draw any of its subsets, it should first tell it to perform a visibility pass by calling the ISpatialTree::ProcessVisibility method. Because this method is a traversal method that is dependant on node type and the number of children each node has, it must be implemented in the derived classes. That is, the quad-tree's version of this method will step into four children at each node while the oct-tree's version will step into eight.

Although this method must be implemented in the derived class, all that is really in the derived class is the traversal logic that determines whether a given node is within the frustum or not and therefore, whether the children should be traversed. For each leaf that is found to have its bounding box inside the camera's frustum, the ProcessVisibility method will issue a call to the leaf's SetVisible method. As we have seen, it is this method that adds the leaf's triangles to render batches in their respective leaf bins. Thus, we have already done the hard work. All that is left to do in the derived class is the tree traversal and frustum tests at each node.

Because the CBaseTree object must make sure that the render batch lists for each leaf bin are reset prior to the visibility traversal taking place, the derived class's ProcessVisibility method must call the CBaseTree::ProcessVisibility method before starting the visibility pass. Below we show the code to the CQuadTree::ProcessVisibility method.

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

    // Start the traversal.
    UpdateTreeVisibility( m_pRootNode, Camera );
}
```

As you can see it is passed (by the application) the CCamera object whose frustum will be used for the visibility pass. It first calls the base class version of the function prior to calling its own UpdateTreeVisibility method. The UpdateTreeVisibility method is the recursive function that traverses the tree and performs the frustum tests at each node. The ProcessVisibility method in the derived classes is really just a wrapper. In fact, the ProcessVisibility method in all derived classes will look identical to this one so we will not show them all. Each derived class's version of this method will first call the CBaseTree::ProcessVisibility method to reset the batch lists in each leaf bin. Then it will call its version of the UpdateTreeVisibility method to perform the visibility traversal starting at the root node. Throughout this section we will only discuss the implementation in the CQuadTree derived class and you can check the source code for the implementations in the other cases.

## ProcessVisibility – CBaseTree

This function is called from the derived class's version of the function to reset the batch lists in each leaf bin. The batch lists are compiled in the CLeafBinData structures in each leaf bin and represent a vertex/index buffer pair. The function loops through each leaf bin and then loops through each CLeafBinData pointer in its internal array. It then resets the CLeafBinData::m_nBatchCount member of each CLeafBinData structure in the leaf bin to zero.

```
void CBaseTree::ProcessVisibility( CCamera & Camera )
{
    LeafBinMap::iterator        BinIterator  = m_LeafBins.begin();
    ULONG                       i;

    // Iterate through the leaf bins and destroy them
    for ( ; BinIterator != m_LeafBins.end(); ++BinIterator )
    {
        CLeafBin * pBin = BinIterator->second;
        if ( !pBin ) continue;

        // For each buffer
        for ( i = 0; i < m_nVBCount; ++i )
        {
            CLeafBinData * pData = pBin->GetBinData( i );
            if ( !pData ) continue;

            // Reset the batch count
            pData->m_nBatchCount = 0;

        } // Next Buffer

    } // Next Leaf Bin

    // Clear the visible leaf list.
    m_VisibleLeaves.clear();
}
```

## 15.4.4 Traversing the Tree to Ascertain Leaf Visibility

It is clear from the previous discussion that the visibility traversal will be performed in the derived class's UpdateTreeVisibility method. In theory, this method could be extremely simple and just traverse through all the nodes in the tree calling the CCamera::BoundsInFrustum method to test if the current node's bounding volume is within the frustum. If it is not, then it calls the node's SetVisible method to traverse down the rest of that tree and set the visible status all leaves below it to false. If the node is inside the frustum (or partially inside the frustum) we can traverse into its children and test their volumes against the frustum in the same manner. Eventually, we will traverse to all leaves that are inside the frustum, at which point the leaf's SetVisible method will be called. As we have seen, this will cause the triangles in those leaves to be added to the render batches in the leaf bins in which they are contained.

While there is certainly nothing wrong with such a strategy, it does not take full advantage of the hierarchical nature of the tree to reduce the number of plane tests that will need to be performed at each node. Furthermore, it does not take into account the fact that if node was found to be outside a frustum plane in the previous visibility update (outside the frustum), because the movement of the camera will typically be very small between frame updates, the node will probably still be outside that frustum plane in the next update. Therefore, when testing the bounding box of a node against the frustum, we should test the plane it failed on in the last update first in the hopes of rejecting the node immediately without having to perform tests for the rest of the frustum planes. This is referred to as frame coherence. It is essentially the concept of using information collected on a previous frame update to optimize the process in the current one. Let us discuss the frame coherence optimization first.

## 15.4.5 Frustum Culling with Frame to Frame Coherence

Figure 15.5 depicts a 2D representation of a node's AABB and the current world space position of the camera's frustum planes. As this is a 2D representation, the frustum has four planes instead of the usual six. We can clearly see that the node is not visible because its bounding box is outside the frustum. This obviously means the node's children are also invisible and there is no need to test them against the frustum. Thus, we can immediately mark this node and all its children as invisible.

The planes of the frustum are labeled 1 through 4 and describe the order in which the frustum planes will be tested against the bounding box.

> **Note:** Frustum culling AABBs was discussed in Module I and will not be discussed again here.

If we walk though the frustum test, we will see that it is only when the polygon box is tested against the fourth plane that we know it is outside the frustum.

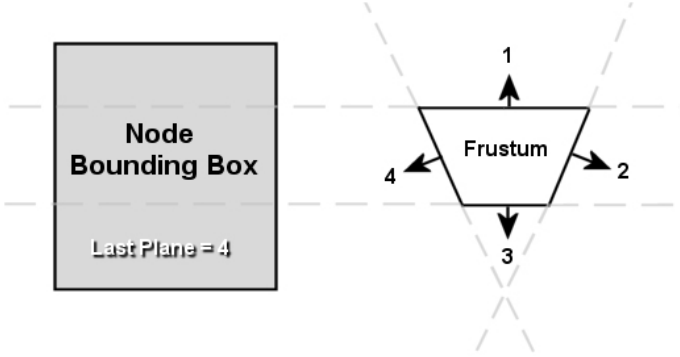



**Figure 15.5**

For example, we can see that when plane 1 is tested, the box is not totally in front of the plane. Since it spans this plane, the box could very well be in the frustum at this point. Next we test the box against frustum plane 2 where we once again discover the box is totally in its backspace and therefore may still be within the frustum. Remember, we only know that an AABB is inside the frustum if the box is behind all planes (plane normals are assumed to be facing outwards in this example). So far we have tested two planes and have not found a plane the box is totally in front of, so next we move on to plane 3. Once again, the box is not totally in front of this plane, so we know at this point that the box is at least partially contained behind the three planes tested and may well be in the frustum. Finally, we test frustum plane 4 where we discover that the box is totally in its front space and therefore could not possibly be visible. At this point we set the visibility status of all leaves underneath that node to false.

Although this works fine, it took us four plane tests to determine that the box was outside. However, if we knew in advance that we would fail against plane 4, we could have rejected the node with one plane test. Although this might sound like a minimal savings at the node level, just remember that your spatial tree might have thousands of nodes, and the 3D frustum will have six planes. In the worst cases, you could end up having to perform six frustum plane tests at a great many nodes.

If we examine Figure 15.5 again we can imagine that during the next frame update, where the camera will likely only be moved by a very small amount (say, vertically), it would fail on the same plane again. So if we know the plane that we failed on last time, which in this case was frustum plane 4, we could store the index of that plane in the node and make sure that when this node is encountered in the next visibility pass, we test plane 4 first. Although our first visibility pass would require that we test all four planes to learn that the fourth plane was the one that caused the rejection, the next visibility pass (next iteration of the game loop) can be potentially optimized when we visit that node, by reading back the index of the plane we failed on last time and making sure that the CCamera::BoundsInFrustum function test this plane first. In a great many cases, this will lead to immediate rejection of large portions of the tree using a single plane test. Again, if a node was outside a certain frustum plane during one frame update, it will probably be outside that same frustum plane during the next frame update (camera changes are generally minimal between frames when an application is running at 30+ frames per second).

We will implement this frame-to-frame coherency in our visibility system. You now know why each of the node classes we developed in the previous lesson stored that signed char member variable called LastFrustumPlane. As a reminder, we show the CQuadTree node class declaration below.

```
class CQuadTreeNode
{
public:

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


    // Public Functions for This Class
    void SetVisible( bool bVisible );

    // Public Variables for This Class
```

```
    CQuadTreeNode * Children[4];        // The four child nodes
    CBaseLeaf     * Leaf;               // If this is a leaf, store here.
    D3DXVECTOR3     BoundsMin;          // Minimum bounding box extents
    D3DXVECTOR3     BoundsMax;          // Maximum bounding box extents
    signed char     LastFrustumPlane;   // The frame coherance 'last plane' index.
};
```

When the node is first created, its LastFrustumPlane index will be set to -1 in the constructor. This is because we do not yet have the index of a frustum plane at which this node failed. It will also be set back to -1 for any node that was found to be visible in the previous visibility update. However, for nodes that are currently outside the frustum, or were found to be outside the frustum in the previous visibility update, this member will store a value between 0 and 5, describing the index of the first frustum plane that generated the outside the frustum result. When this node is visited in the next visibility update, and the CCamera::BoundsInFrustum method is called to test the bounding box of the node against the frustum planes, we will pass this plane index into that function to instruct it to test this plane first. Hopefully, the BoundsInFrustum method will immediately determine that the node is still outside this plane and will return the invisible status. If the camera has been moved from the previous position and the node is no longer totally in front of this plane, we will just perform a test on the rest of the planes as normal. If one of the other planes is found to have the AABB totally in its front space, the node's LastFrustumPlane member is updated to store the index of that plane. If the box is not in front of any planes and is therefore inside (or partially inside) the frustum, the node's LastFrustumPlane member is set to -1 and a visible status is returned for the node. The next time this node is encountered in the next visibility process, it will have -1 in its LastFrustumPlane member which means the BoundsInFrustum test will just loop through and test all 6 planes in the usual way.

In a moment we will see how the CCamera::BoundsInFrustum method has been updated to test against the chosen frustum plane first if such a plane index is passed into the function. We will also see how it will update the value of the node's LastFrustumPlane member if a new 'first' rejection plane is found. But before we discuss the code for the frame-to-frame coherence optimization, we will talk about a second optimization that can be introduced to make use of the spatial hierarchy and further reduce the number of frustum plane tests that have to be performed at each node.

## 15.4.6 Hierarchical Frustum Culling

In the previous chapter we discussed hierarchical frustum culling as the process of stepping through the nodes of the tree and rejecting any node (and all its children) that is outside the frustum from our rendering pipeline without further testing. However, we can further exploit the parent/child relationship of nodes in the tree to introduce another means for frustum plane test reduction in the typical case.

This technique is used to speed up the case where the node is not totally inside or outside the frustum, but is intersecting in such a way that the node's volume is totally behind/inside one or more of the frustum planes. Although we know that in the intersecting case, the children of the node must also be tested against the frustum, we also know (because of the child/parent relationship) that if the parent node is totally behind/inside one of the frustum planes, all of its child nodes/leaves will be as well. Therefore,

when testing the children, we no longer need to test this plane against their volumes because we know that they will always be found to be in the plane's back space.

Figure 15.6 demonstrates this idea in two dimensions by showing a parent node's AABB and the AABBs of its four immediate children.

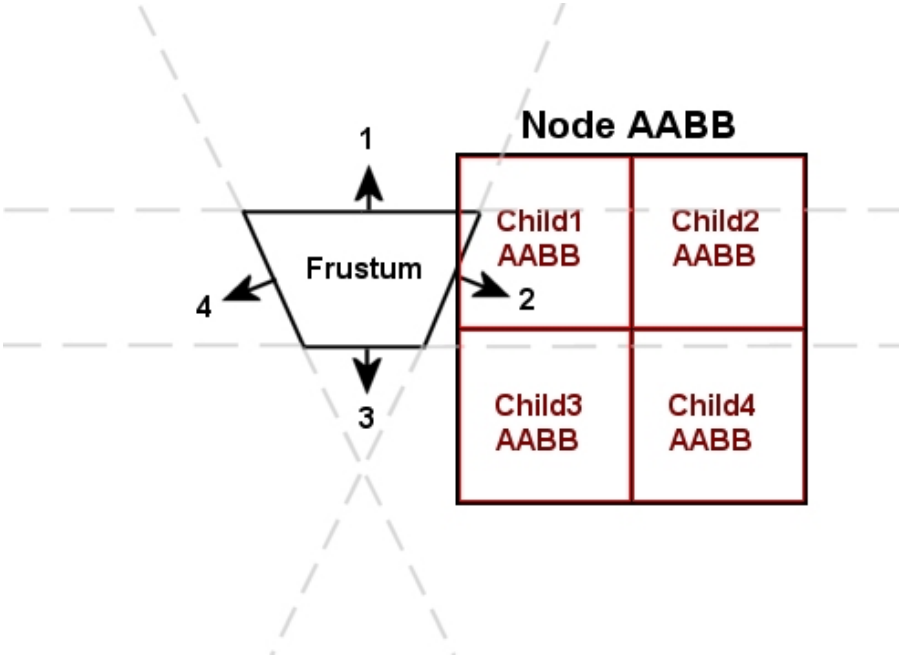



**Figure 15.6**

The important frustum plane to look at in this example is frustum plane 4. We can see that the parent node's volume intersects frustum planes 1, 2, and 3 and is totally behind frustum plane 4. This tells us that part of the parent node is behind all frustum planes so the node is visible. As the node is not completely inside the frustum, this does not mean that we should set all child nodes to visible automatically. We can see for example, that although the parent node intersects the frustum, three of its children do not. Indeed they are fully outside the frustum. Therefore, if the frustum is intersecting the node, the child nodes must also be tested to determine their visibility status. This means that in a worst case scenario, we have to do six frustum plane tests against the node's volume and perform the same six tests for each of its children, and so on.

If we look at frustum plane 4 in Figure 15.6 we can see that the parent node is completely behind (inside) this plane. But because the parent node is behind plane 4, so are all of its children. This is clear from just looking at the diagram. So we recognize that there is no need to test this plane against the children. The parent can essentially say to the child nodes, "I am behind this plane, so all of you are too. Do not bother performing a test against this plane; just assume that you are behind it and test the other planes instead".

The recursive visibility process will pass an array of bits from parent to child as it traverses the tree. Each bit will represent one of the six frustum planes. The CCamera::BoundsInFrustum function will be updated to take an unsigned char whose first six bits will be used to represent the status of the frustum planes. This unsigned char will be set to zero at the beginning of the traversal.

When a node is reached, its bounding box will be passed into the CCamera::BoundsInFrustum function along with this unsigned char. When this function finds any plane that the AABB of the node is completely behind, it will set the bit to 1 in the unsigned char corresponding to that plane. When the function returns back to the node, the node flag will have the corresponding bits set to 1 for any planes the node's volumes is totally behind. This unsigned char is then passed down to the children. The child

nodes will then pass this unsigned char of frustum plane bits into their CCamera::BoundsInFrustum calls so that that any planes at this node which result in the fully behind case can also have their bits set to 1. The important point here however, is that the CCamera::BoundsInFrustum test will only test planes which have not had their bits set to 1 in the passed unsigned char.

Now that we know the two optimizations we will apply during the frustum culling traversal of the tree, we are ready to look at an example of the UpdateTreeVisibility method. Since these methods are the same in all derived classes (with the exception of having to step into a varying number of children), we will only show the CQuadTree::UpdateTreeVisibility method here.

## UpdateTreeVisibility - CQuadTree

This method is called from the CQuadTree::ProcessVisibility function to start the recursive process for the root node. It repeatedly calls itself until all leaves in the tree which intersect the camera's frustum have there visible status set. The function is very simple as nearly all the optimizations are performed inside the modified CCamera::BoundsInFrustum method, which we will discuss in a moment.

The function is passed three parameters. The first is the node currently being visited by the function. This will be the root node of the tree when this function is first called from the CQuadTree::ProcessVisibility method. The second parameter is the camera whose frustum will be used for the visibility test. The third parameter is the function's means of passing the array of 'Totally Inside' frustum bits from one node to the next. Since this parameter is not passed for the root node, the value of the FrustumBits unsigned char will be set to 0 when tree traversal begins at the root.

The first thing the function does is call the CCamera::BoundsInFrustum method, passing in the node's bounding box extents, the unsigned char of frustum plane bits, and the node's LastFrustumPlane member.

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

The third parameter to this function is an optional world matrix that can be used to transform the input bounding box into world space prior to performing the test. As our node's bounding box is already in world space, no transformation is required so we set the matrix pointer to NULL.

Before examining the rest of this function, let us just make sure we understand what may have happened when the above function has returned. Any bits set to 1 in the passed FrustumBits char will not be tested

against the frustum. Instead, the function will assume that the node's box is totally behind these planes and will progress to the next plane that needs to be tested. If any of the planes that did require testing (their bits were set to 0) are found to contain the bounding box completely in its back space, that plane will have its bit set to 1 in the FrustumBits char and will also not need to be tested again for any children of this node. Finally, if the BoundsInFrustum method did determine that the node is outside the frustum, the LastFrustumPlane member of the node will have its value altered such that it contains the index of the first plane it failed on. This will be the first plane that is tested by this function the next time the BoundsInFrustum function is called for this node.

The BoundsInFrustum method will return one of three values which are members of the FRUSTUM_COLLIDE enumerated type defined inside the CCamera namespace. This enumerated type is shown below with an explanation of its members.

```
enum FRUSTUM_COLLIDE
{
        FRUSTUM_OUTSIDE     = 0,
        FRUSTUM_INSIDE      = 1,
        FRUSTUM_INTERSECT   = 2,
        FRUSTUM_FORCE_32BIT  = 0x7FFFFFFF
};
```

**FRUSTUM_OUTSIDE**
The node's bounding volume is totally outside the frustum (the node is not visible). This means we will not perform any more frustum tests down this branch of the tree and we can immediately set the visibility status of any leaves stored under this node to false. If the parent node is completely outside the frustum, so too must be all of its children.

**FRUSTUM_INSIDE**
The node's bounding volume is contained completely within the frustum (the node is visible). This also means that we do not have to perform any more frustum tests down that branch of the tree and we can immediately set the visibility status of all leaves stored under that node to true. If the parent node is fully contained within the frustum, so too must be all of its children.

**FRUSTUM_INTERSECT**
The node's bounding volume is partially contained within the frustum (the node is visible). This means we must traverse into the children and continue to perform frustum tests for each child. If the parent node is partially intersecting the frustum, one of more of its children will be visible.

Now that we know what the results mean, let us see the code that processes them in a switch statement.

```
    // Test result of frustum collide
    switch ( Result )
    {
    case CCamera::FRUSTUM_OUTSIDE:
        // Node is not at all visible
        pNode->SetVisible( false );
        return;
```

If the frustum test returned FRUSTUM_OUTSIDE then the node's bounding volume is completely outside the frustum, so the node is not visible and neither are any of its children. When this is the case we simply call the node's SetVisible method passing a visibility parameter of false. This method performs no frustum tests and simply traverses the rest of the branch setting the visible status of any leaves found there to false. The function then returns so that the rest of this branch of the tree avoids further frustum testing.

If the frustum test returned FRUSTUM_INSIDE then the node's bounding volume is entirely contained inside the frustum and therefore, this node and all child nodes are visible and no further frustum tests need to be done down this branch of the tree. Instead we just call the node's SetVisible method passing a visibility status of true. This function will quickly traverse to find all the leaf nodes underneath this current node and will set their visibility status to true. We then return from the function so that we do not perform any more frustum tests down this branch of the tree.

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

Finally, if the frustum test returned FRUSTUM_INTERSECT it means that the current node is visible but some of its children might not be, so we will need to perform further frustum tests along this branch of the tree. If the current node is a leaf, we set its visible status to true (which we know will cause its triangles to be added to their leaf bin's render batch lists).

```
    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
```

Remembering that the function will have returned already in any case other than FRUSTUM_INTERSECT, this last section of code is also only executed in the intersection case. It simply loops through each child node and recurs into it.

```
    // The remaining case (FRUSTUM_INTERSECT) means we need to test further
    for ( i = 0; i < 4; ++i )
    {
        if ( pNode->Children[i] ) UpdateTreeVisibility( pNode->Children[i],
                                                        Camera,
                                                        FrustumBits );

    } // Next Child
}
```

You will find that the UpdateTreeVisibility method in all the derived classes will be identical to the code shown above, with the exception of the number of children traversed.

## SetVisible - CQuadTreeNode

Although we took a brief look at this method in the previous lesson, we will take a quick look at it again now that we have seen it being called in the FRUSTUM_INSIDE and FRUSTUM_OUTSIDE cases in the above function.

```
void CQuadTreeNode::SetVisible( bool bVisible )
{
    unsigned long i;

    // Set leaf property
    if ( Leaf  ) { Leaf->SetVisible( bVisible ); return; }

    // Recurse down if applicable
    for ( i = 0; i < 4; ++i )
    {
        if ( Children[i] ) Children[i]->SetVisible(bVisible);

    } // Next Child
}
```

As you can see, if the node is a leaf it calls its SetVisible method to instruct it to add its polygon data to the render batches in each leaf bin. As we now know, this will also instruct the leaf to add its ‘this’ pointer to the tree’s visible leaf list.

Of course, the final piece of the puzzle is the CCamera::BoundsInFrustum method. Although this method has been in use since Module I in this series, we have now modified it to accept some additional parameters that optimize the frustum rejection for a spatial tree. After we have look at the code to that function, we will have covered all of the code that is executed when the application calls the ISpatialTree::ProcessVisibility method. Thus we will have completely explored how the visibility pass on the tree is performed.

## BoundsInFrustum - CCamera

The frustum culling of AABBs and the extraction of the frustum planes in world space was explained in detail in Module I. This function has been with us for quite some time now and we have seen it used in nearly all our lab projects since its inception. Therefore we will not be covering how frustum culling works when discussing this code. If you do not remember how the frustum culling of AABBs can be done, please refer back to Chapter 4 in Module I for the details.

The first thing this function does is call the CCamera::CalcFrustumPlanes method to extract the world space planes for the view frustum and store them in the camera object’s m_Frustum planes array (which

contains 6 elements). We then make a copy of the passed FrustumBits unsigned char into the nBits local variable for ease and speed of access. We also copy the value stored in the passed LastOutside char which will contain the index of the plane which the node that called this function failed on the last time this function was called.

```
CCamera::FRUSTUM_COLLIDE CCamera::BoundsInFrustum( const D3DXVECTOR3 & vecMin,
                                                   const D3DXVECTOR3 & vecMax,
                                                   const D3DXMATRIX * mtxWorld ,
                                                   UCHAR * FrustumBits ,
                                                   signed char * LastOutside )
{
    // First calculate the frustum planes
    CalcFrustumPlanes();

    ULONG           i;
    D3DXVECTOR3     NearPoint, FarPoint, Normal, Min = vecMin, Max = vecMax;
    FRUSTUM_COLLIDE Result  = FRUSTUM_INSIDE;
    UCHAR           nBits = 0;
    signed char     nLastOutside = -1;

    // Make a copy of the bits passed in if provided
    if (FrustumBits) nBits = *FrustumBits;

    // Make a copy of the 'last outside' value to prevent us having to dereference
    if ( LastOutside ) nLastOutside = *LastOutside;
```

If the caller passed a matrix into the function, then it means we were passed a model space bounding box which should be transformed into world space using this matrix before the frustum planes are tested. That is no problem because we wrote a function in the previous lesson that did exactly that.

```
    // Transform bounds if matrix provided
    if ( mtxWorld ) MathUtility::TransformAABB( Min, Max, *mtxWorld );
```

If the node's LastFrustumPlane value (now stored in nLastOutside) is not set to -1, it contains a valid index for a frustum plane and we will test that plane first. However, we will only test it if that plane does not have its bit set to 1 in the nBits array. If it does, then regardless of the fact that the node was found to be outside the plane in the previous visibility pass, it must be inside it now, because one of its parent nodes higher in the tree was found to be completely inside it and set this bit to 1. Alternatively, if a valid last plane index is passed which does not currently have its bit set, we will perform the test.

As the nLastOutside variable contains the index of the plane we want to test first, we will extract the plane normal from the camera's m_Frustum array. This array stores the planes in <a,b,c,d> format so we know that the normal is contained in members a, b, and c.

```
    // If the 'last outside plane' index was specified, test it first!
    if ( nLastOutside >= 0 && ( ((nBits >> nLastOutside) & 0x1) == 0x0 ) )
    {
        // Store the plane normal
        Normal = D3DXVECTOR3( m_Frustum[nLastOutside].a,
                              m_Frustum[nLastOutside].b,
                              m_Frustum[nLastOutside].c );
```

We then use the plane normal to calculate the near and far points on the bounding box with respect to the plane.

> **Note:** Remember from our Plane/AABB intersection discussion that the near point is the point that would intersect the plane first were it located totally in the plane's front space and slowly moved towards the plane until the point of intersection. The far point the last point on the AABB that would cross the plane in the same scenario.

```
        // Calculate near / far extreme points
        if ( Normal.x > 0.0f ) { FarPoint.x  = Max.x; NearPoint.x = Min.x; }
        else                   { FarPoint.x  = Min.x; NearPoint.x = Max.x; }

        if ( Normal.y > 0.0f ) { FarPoint.y  = Max.y; NearPoint.y = Min.y; }
        else                   { FarPoint.y  = Min.y; NearPoint.y = Max.y; }

        if ( Normal.z > 0.0f ) { FarPoint.z  = Max.z; NearPoint.z = Min.z; }
        else                   { FarPoint.z  = Min.z; NearPoint.z = Max.z; }
```

If the near point is found to be in front of the plane, the entire box must be in front of the plane. Consequently, if the box is totally in front of any of the planes, then it must be completely outside the frustum. As soon as such a plane is found, we return FRUSTUM_OUTSIDE

> **Note:** Our frustum planes normals face outwards.

```
        // If near extreme point is outside, then the AABB is totally outside
        if ( D3DXVec3Dot( &Normal, &NearPoint ) + m_Frustum[nLastOutside].d>0.0f )
           return CCamera::FRUSTUM_OUTSIDE;
```

If we have not returned from the function yet, it means the near point is in the backspace of the plane, so we next test to see if the far point is in the front space. If it is, the box is spanning the frustum plane and we return FRUSTUM_INTERSECT.

```
        // If far extreme point is outside, then the AABB is intersecting
        if ( D3DXVec3Dot( &Normal, &FarPoint ) + m_Frustum[nLastOutside].d > 0.0f )
            Result = CCamera::FRUSTUM_INTERSECT;
```

If we have still not returned from the function, it must mean that both the near and far points of the box are behind the frustum plane. This means the box is completely contained in the backspace of the plane and we must test the rest of the frustum planes. However, if the box is completely contained in the backspace of this plane, it also means all of the node's children will be too. Therefore, when this function is called for the child nodes, we should skip testing this plane and assume that they are in the backspace also. To do this, we set the bit for this plane in the nBits char to update the bit set that is passed down to the child nodes. We set the bit that corresponds to this plane by shifting the value 1 by the appropriate number of bits to the right (the number of bits to shift is equal to the index of the plane we are setting the bit for) and OR'ing it with the current bit set.

```
        else
            nBits |= (0x1 << nLastOutside); // We were inside, update our bit set
```

```
    } // End if last outside plane specified
```

If we reach this part of the function, it means that either no valid LastFrustumPlane value for the node was passed or that it was tested first, but the node is now found to be intersecting or behind that plane.

In the next section we simply create a loop from 0 to 6 to loop through the 6 frustum planes so that we can test the rest of them. We will skip a plane if its bit is already set to 1 in the nBits array and also skip the plane that was tested first in the above section of code (whose index is contained in nLastOutside).

```
    // Loop through all the planes
    for ( i = 0; i < 6; i++ )
    {
        // Check the bit in the uchar passed to see if it should be
        // tested (if it's 1, it's already passed)
        if ( ((nBits >> i) & 0x1) == 0x1 ) continue;

        // If 'last outside plane' index was specified,
        // skip if it matches the plane index
        if ( nLastOutside >= 0 && nLastOutside == (signed char)i ) continue;
```

If we get this far in the loop code, it means the current plane being processed has not yet had its bit set to 1 in the frustum bits char and it is not the plane we have already tested. As with the first plane we tested, we extract the normal of the current plane and calculate the near and far points.

```
        // Store the plane normal
        Normal = D3DXVECTOR3( m_Frustum[i].a, m_Frustum[i].b, m_Frustum[i].c );

        // Calculate near / far extreme points
        if ( Normal.x > 0.0f ) { FarPoint.x  = Max.x; NearPoint.x = Min.x; }
        else                   { FarPoint.x  = Min.x; NearPoint.x = Max.x; }

        if ( Normal.y > 0.0f ) { FarPoint.y  = Max.y; NearPoint.y = Min.y; }
        else                   { FarPoint.y  = Min.y; NearPoint.y = Max.y; }

        if ( Normal.z > 0.0f ) { FarPoint.z  = Max.z; NearPoint.z = Min.z; }
        else                   { FarPoint.z  = Min.z; NearPoint.z = Max.z; }
```

Next we calculate the distance from the near point to the plane. If it is found to be in the frontspace of the plane, we know the entire box must be outside the frustum so we can return FRUSTUM_OUTSIDE. However, before we return, we also store the index of this plane in the node's LastFrustumPlane member (pointed to by LastOutside) so that this function will test this plane first when the node is frustum tested again in the next visibility pass. We also copy the contents of the nBits char which currently contains all the frustum planes that the node is totally inside of. This is assigned to the FrustumBits pointer so that it is returned from the function to the node, where it can be passed down to its children.

```
        // If near extreme point is outside, then the AABB is totally outside
        // the frustum
        if ( D3DXVec3Dot( &Normal, &NearPoint ) + m_Frustum[i].d > 0.0f )
```

```
        {
            // Store the 'last outside' index
            if ( LastOutside ) *LastOutside = (signed char)i;

            // Store the frustm bits so far and return
            if (FrustumBits) *FrustumBits = nBits;
            return CCamera::FRUSTUM_OUTSIDE;

        } // End if outside frustum plane
```

If the near point is not in front of the plane but the far point is, it means the box is spanning the plane so we return FRUSTUM_INTERSECT.

```
        // If far extreme point is outside, then the AABB is intersecting
        // the frustum
        if ( D3DXVec3Dot( &Normal, &FarPoint ) + m_Frustum[i].d > 0.0f )
            Result = CCamera::FRUSTUM_INTERSECT;
```

Otherwise, it means the current plane being processed has the box contained totally in its backspace. Since this means that all of the node's children will also share this relationship with the plane, we set the bit that corresponds to this plane so that the children know they do not have to process it.

```
        else
            nBits |= (0x1 << i);
           // We were totally inside this frustum plane, update our bit set

    } // Next Plane
```

If we have not yet returned from the function it means the box is inside the frustum. Since there was no plane that rejected it, we set the node's last plane index (pointed to by LastOutside) to -1. We then copy the nBits array into the FrustumBits parameter so that they are accessible to the caller on function return. We then return a result of FRUSTUM_INSIDE.

```
    // Store none outside
    if ( LastOutside ) *LastOutside = -1;

    // Return the result
    if (FrustumBits) *FrustumBits = nBits;
    return Result;
}
```

We have now seen all of the code involved in the visibility processing procedure for the tree. This is all invoked when the application calls the ISpatialTree::ProcessVisibility function. In the last and final section covering the CBaseTree rendering system, we will examine how the visible data is rendered.

## 15.4.7 Rendering the Visible Static Polygon Data

After the application has called the ISpatialTree::ProcessVisibility method to flag the visible leaves and build the render batches in each leaf bin, all it has to do is loop through each subset in use by the scene and call the ISpatialTree::DrawSubset method. This method is shown below and is the final function of CBaseTree we will cover that pertains to its internal rendering system.

### DrawSubset - CBaseTree

This method is called by the application to draw a particular subset in the tree. The application will typically set the texture and material required to render this subset prior to making this call. This function essentially just calls other functions which we have already covered.

Provided the device is valid, we use the CBaseTree::GetLeafBin method to fetch a pointer to the leaf bin for the associated subset ID. If a leaf bin does not exist for this subset ID, it means the tree contains no polygon data that uses this subset. In such situations, the GetLeafBin function returns NULL and we return from the function without taking any further action.

> **Note:** Since the scene is using global attribute IDs for all objects, it is entirely possible that the tree's polygon data will only use a handful of those attributes. As the application will essentially call DrawSubset for each global attribute, this function may be called many times with an attribute ID for which no polygon data exists in the tree for and for which no leaf bin has been created.

```
void CBaseTree::DrawSubset( unsigned long nAttribID )
{
    // Can draw?
    if ( !m_pD3DDevice ) return;

    // Retrieve the applicable leaf bin for this attribute
    CLeafBin * pLeafBin = GetLeafBin( nAttribID );
    if ( !pLeafBin ) return;

    // Render the leaf bin
    pLeafBin->Render( m_pD3DDevice );
}
```

After a valid leaf bin has been retrieved, its Render method is called. We looked at this function earlier and saw how it rendered all the render batches for each index/vertex buffer combination it contains triangles for.

# 15.5 Rendering the Tree – Application Perspective

Rendering the tree's static geometry could not be easier. Below we see the CScene::Render method from Lab Project 14.1. Because this function is now getting quite large (given all of the code to set up render states, enables lights and fog, rendering the CObject array, etc.), we have snipped a good amount of the code out to condense the listing.

You will recall that the first part of this function sets up certain render states, renders the sky box and enables lighting and fog. In this lab project we have added a new function call to the ISpatialTree::ProcessVisibility method at the bottom of the next section of code.

```
void CScene::Render( CCamera & Camera )
{
    ULONG i, j;
    long  MaterialIndex, TextureIndex;

    if ( !m_pD3DDevice ) return;

    //……SNIP : Set up Global Render states here

    // Render the skybox first !
    RenderSkyBox( Camera );

    //……SNIP : Set up lights and fog here……

    // Allow the spatial tree to process visibility
    m_pSpatialTree->ProcessVisibility( Camera );
```

Now that the visibility status of the tree has been updated using the camera's current position and orientation, we set the device world matrix to an identity matrix because the static geometry stored in the tree is already in world space.

```
    // Loop through each scene owned attribute
    D3DXMATRIX mtxIdentity;
    D3DXMatrixIdentity( &mtxIdentity );

    m_pD3DDevice->SetTransform( D3DTS_WORLD, &mtxIdentity );
```

Now it is time to render the subsets of the spatial tree. We do this by looping through the array of attributes used by the scene. For each attribute, we set its texture and material on the device and call the ISpatialTree::DrawSubset method, passing in the respective attribute ID. If the tree has any data for this attribute, its associated leaf bin will render its triangles.

```
    for ( j = 0; j < m_nAttribCount; j++ )
    {
        // Retrieve indices
        MaterialIndex = m_pAttribCombo[j].MaterialIndex;
        TextureIndex  = m_pAttribCombo[j].TextureIndex;
```

```
        // Set the states
        if ( MaterialIndex >= 0 )
            m_pD3DDevice->SetMaterial( &m_pMaterialList[ MaterialIndex ] );
        else
            m_pD3DDevice->SetMaterial( &m_DefaultMaterial );

        if ( TextureIndex >= 0 && m_pTextureList[ TextureIndex ] )
            m_pD3DDevice->SetTexture( 0, m_pTextureList[ TextureIndex ]->Texture );
        else
            m_pD3DDevice->SetTexture( 0, NULL );

        // Render all faces with this attribute ID
        m_pSpatialTree->DrawSubset( j );

    } // Next Attribute
```

At this point the tree has been rendered in its entirety, so next we loop though the scene's CObject array (which will now store only the dynamic objects in use by the scene, like CActors) and render each one. We will discuss how this works in the next section. Then we can iterate through our terrain array and render any terrains in use. We do not show the code for this as it has not changed from previous demos (we will cover dynamic object rendering in the next section).

Finally, our CGameApp class now includes a new member called m_bDebugDraw. This is a Boolean that has its state toggled by a 'Debug Draw' menu option. The CGameApp::GetDebugDraw method returns the state of this Boolean. As you can see in the following code, if this function returns true, it means the user would like to enable debug drawing and thus the spatial tree's DebugDraw method is called. We discussed the code to this function in the previous lesson and saw how it traverses the tree and draws the bounding boxes at each node.

```
    // …… SNIP : Render all CObjects here (dynamic actors and tri mehses)

    // …… SNIP : Render any terrains here

    // Draw the spatial tree's debug data if requested
    if ( GetGameApp()->GetDebugDraw() ) m_pSpatialTree->DebugDraw( Camera );

}
```

And there we have it. That is all of the code involved in render our spatial tree from the application's perspective. All it took was two function calls to ISpatialTree member functions.

We have now covered the complete rendering system used by CBaseTree. In the end, we have built ourselves a very useful set of spatial managers that optimize both collision queries and scene rendering. In the final section of this chapter we will examine how our dynamic objects can also benefit from the visibility information in our tree. This will enable us to render only the actors that are currently considered visible by the tree.

# 15.6 Managing Dynamic Objects

So far we have implemented spatial trees that compile static polygon data and we have examined the methods used to prepare and render that data in an efficient manner. We have also added the concept of linking external objects to the leaves of the tree using the concept of detail areas. You will recall that a detail area is a bounding volume that helps shape the tree as it is being built. It is assigned during the compilation phase to the leaves in which it is fully or partially contained. Because each detail area compiled into the tree can also store a context pointer, they allow us to store any type of external object in the tree. A detail area may represent a series of render states, for example, that must be set on the device when the player enters the same leaves as the detail area. This might be handy if you wanted to use detail areas to represent regions of fog within the level or to represent an area where lights should be enabled/disabled. A detail area might also be used to represent a mesh or an actor that we do not want compiled into the tree at the per-polygon level, although we still want it to use the tree's visibility system to inform the application if that object should be rendered. The tree has no understanding of what a detail area represents internally. It just sees it as a bounding volume that has its pointer stored in the leaves in which it is contained. However, the application will know exactly what that detail area represents as it was responsible for registering it with the tree and assigning its context pointer. This context pointer will likely point to some structure or object that has meaning to the application.

The application is ultimately responsible for when and how to process detail areas. If we think about a detail area that represents an external mesh for example, the application would be responsible for rendering that mesh. However, if the detail area is not in any currently visible leaves, the application then knows it does not need to be rendered. In this situation the application can simply query the tree for a list of visible leaves and process only the detail areas in those leaves.

As useful as detail areas are, they do not provide us with coverage for all situations. Since they are compiled directly into the tree, they are totally static concepts. We can link an external mesh or actor to the tree at compile time by registering it as a detail area, but if the application plans on animating or updating the position of that object, detail areas will not suffice. Therefore, a system will have to be added to our tree that works in a similar manner to the detail area idea, but allows for an entity to have its position within the tree dynamically updated.

We will only need to add a handful of methods to CBaseTree in order for our trees to support dynamic objects. The system we will use to register dynamic objects with the tree will be similar to the system we used to register dynamic objects with the collision system. The ISpatialTree::InsertTreeObject method will be used for this purpose. It will store the object internally in the tree and return an ID (a handle) back to the caller. The caller can then use this handle to query for information about that object (such as what leaves is it currently in).

## 15.6.1 The TreeObject Structure

When a dynamic object is registered with the tree, a TreeObject structure is used as the transport mechanism to pass information about the object to the InsertTreeObject method. This structure is defined in ISpatialTree.h and is shown below.

***Excerpt from ISpatialTree.h***

```
typedef struct _TreeObject
{
    void          * pContext;
    bool          * pbVisible;
    long            nTreeObjectIndex;
} TreeObject;
```

When we pass a structure of this type into the InsertTreeObject method, only the first two members are used to describe the object. On function return, the third member (nTreeObjectIndex) will contain the handle assigned to the object by the spatial tree. The application can then store this returned handle in the object and use it for later querying.

**void *pContext**
This member is a pointer to some object that the application would like associated with the dynamic object in the tree. This could be a pointer to a CActor object for example. In our lab project code, we will pass a pointer to a CObject, which will contain either a dynamic CTriMesh or CActor object.

**bool *pbVisible**
This member allows the application to store a Boolean pointer that the tree will automatically update if the object is inside a visible leaf. That is, during the visibility pass, if a leaf is encountered that is visible, any tree objects (dynamic objects) that have been assigned to that leaf will have their Booleans set to true. This provides an alternative way for the application to query the visibility status of a dynamic object rather than fetching the visible leaves and searching them for the object currently being processed.

In Lab Project 14.1, we extend our CObject structure to store a Boolean member called m_bVisible. We will pass a pointer to this Boolean in this member of the TreeObject structure when we register the CObject with the spatial tree. Later, when the ProcessVisibility process is performed and a leaf which contains this object is found to be visible, the value of this Boolean will automatically be set to true by the tree even though it is stored in a CObject structure which has nothing to do with the tree. When rendering dynamic objects, the application can just loop through each one and only render it if the tree has set its Boolean visibility status to true (more on this later)

**long nTreeObjectIndex**
This is not an input parameter to ISpatialTree::InsertTreeObject; it will be set on function return so that the application can retrieve and store this value in the CObject structure. Just like our collision system, this is a unique ID that the tree has given to this pair of context and Boolean pointers inside its internal arrays.

## 15.6.2 The CObject Structure

Our CObject structure has been updated to include two new members. The new version of this structure is shown below with the new members highlighted in bold.

```
class CObject
{
public:

      // Constructors & Destructors for This Class.
         CObject( CTriMesh * pMesh  );
         CObject( CActor   * pActor );

             CObject( );
    virtual ~CObject( );


    // Public Variables for This Class
    D3DXMATRIX                  m_mtxWorld;
    CTriMesh                   *m_pMesh;
    CActor                     *m_pActor;
    LPD3DXANIMATIONCONTROLLER   m_pAnimController;
    CActionStatus              *m_pActionStatus;
    long                        m_nObjectSetIndex;

    long                        m_nTreeObjectIndex;
    bool                        m_bVisible;            // Is this object visible?
};
```

**m_nTreeObjectIndex**
This member will contain the handle (unique ID) assigned to the object by the spatial tree in response to the ISpatialTree::InsertTreeObject method being called for this object. Whenever we wish to update the position of the object or remove it from the spatial tree, it is this handle that we pass into the ISpatialTree::UpdateTreeObject and the ISpatialTree::RemoveTreeObject methods so that the tree knows exactly which tree object we are referring to.

**m_bVisible**
This member is a Boolean that describes the visibility status of the object. The address of this Boolean will be passed into the ISpatialTree::InsertTreeObject method (via the TreeObject structure) when the object is first registered. The spatial tree will automatically set this Boolean to true when the object is found to be in a visible leaf. The scene now has visibility information for the external object automatically with having to intersect with the tree itself. The tree will automatically set the m_bVisible Booleans of any CObjects in use by the application that have been registered with the spatial tree and are currently in visible leaves.

Before we discuss the ISpatialTree management of dynamic objects, we will take a look at how the application registers and updates the positions of dynamic objects using the ISpatialTree member functions. This will give us a better understanding of the system we need to implement and the way our application will expect that system to behave.

## 15.6.3 Registering a Dynamic Object with the Spatial Tree

As was the case with our collision system, any dynamic object that we create is registered with the spatial tree and returned a unique ID for that object within the system. In Lab Project 14.1 we load our scenes from IWF files and, as we have seen, the static geometry stored in the IWF file is added to the tree as static polygon data inside the CScene::ProcessVertices function. We have also seen in nearly all previous lab projects, that our dynamic objects are usually stored in the IWF file as external references to X files which are loaded into actors and stored in the scene's CObject array. It is the ProcessReference method that has always been the function used to create and load such X files into actors. It is in this function that the newly created CObject structure is registered with our collision system. It is also the function where we will register the object with the spatial tree.

The following snippet of code has been added to the very bottom of the CScene::ProcessReference function. It is responsible for adding the newly created CObject (which at this point may store a pointer to a CActor or a CTriMesh) with the spatial tree. In this code, which is only executed if the scene has a spatial tree, the new CObject which has been created earlier in the function (and populated accordingly) has been assigned to the pNewObject pointer.

```
// Add to the spatial tree's object list if applicable
if ( m_pSpatialTree )
{
  TreeObject Object;
  Object.pbVisible = &pNewObject->m_bVisible;
  Object.pContext  = NULL;

  // Add to the tree
  pNewObject->m_nTreeObjectIndex = m_pSpatialTree->InsertTreeObject( Object );

} // End if has spatial tree
```

As you can see, a new TreeObject structure is instantiated as our means of describing the dynamic object to the tree. The TreeObject's Boolean pointer is assigned to point at the m_bVisible Boolean member in our CObject structure for later updates during visibility testing. Notice how we set the context pointer of the TreeObject structure to NULL as we do not need to use it. You could assign this to point at the CObject structure itself, which would be useful if your means for determining dynamic object visibility was to loop through the currently visible leaves searching for TreeObjects that are then rendered. However, our current application only needs to know whether any of the CObjects in its array need to be rendered. Since we have registered the object's Boolean, this will be adequate for our purposes. All our application will need to know when looping through the CObject array is which ones need to be rendered. How you choose to do it is up to you as both methods suit different situations. Using the Boolean means that the application is not required to perform queries into the tree to determine visibility status.

After we have set up the TreeObject structure we then call ISpatialTree::InsertTreeObject passing in TreeObject structure so that this object can be stored in the tree's dynamic object array. The return value from this function is the handle (ID) of the new dynamic object we have just added which has been assigned by the spatial tree. We store this in the CObject's m_nTreeObjectIndex member so that when

our application alters the position of this object in the scene, it can pass this ID into the ISpatialTree::UpdateTreeObject to inform the tree that the object has been updated and that the leaves in which it is currently contained need to be recalculated.

Note that all we have done here is register a single Boolean pointer with the tree. We have not even assigned the object a bounding volume. So how can the tree know which leaves the object should be assigned to? The answer is that it does not; at least not currently. All the above function does is create an entry in the tree's dynamic object array where the passed Boolean pointer and context data pointer are stored. At this point, it is not assigned to any leaves. This is what the ISpatialTree::UpdateTreeObject method is for.

## 15.6.4 Updating Dynamic Tree Objects

When the application updates the position of an object (i.e., changes its world matrix), we must inform the tree so that it can remove that object's pointer from any leaves in which it is currently contained and assign it to the appropriate new leaves. This is all done automatically when the application makes a call to the ISpatialTree::UpdateTreeObject function. It is this function that is passed the ID of the object we would like to update and a world space AABB. This function will first remove the object from any leaves in which it is currently assigned. It will then pass the AABB of the object down the tree (using the CollectLeavesAABB method) to collect a list of leaves in which the bounding box is contained. The object then has its pointer added to each leaf in which it is contained. This same list of leaves is then stored inside the tree in a structure that pairs the tree object with its intersected leaf list.

Each leaf in our tree will now potentially store a list of TreeObject pointers and each TreeObject stored in the tree will also be stored alongside a list of leaves in which the object is currently contained. This provides yet more convenient ways to use dynamic objects with the system. As each tree object is stored in an array paired with a list of leaves in which it is currently contained, this means the application can quickly access the leaf list for an object using the ISpatialTree::GetTreeObjectLeaves method. This method is passed the ID of a tree object and will simply look up that object in the tree's internal array and return the leaf list that is stored alongside it. Additionally, because the leaves themselves also store a list of tree object pointers describing the list of dynamic objects contained within them, the application can call the ISpatialTree::GetVisibleLeafList method and then parse the returned visible leaves to see if the object in which it is interested is stored there. These few methods provide the application with choices about the way it would like to work with and query the status of its dynamic objects.

The following section of code is taken from the CScene::AnimateObjects method in Lab Project 14.1. This function is certainly familiar to us at this point, but now a few new lines have been added so that the spatial tree is informed about object position updates so that it can rebuild the leaf lists for that object.

The first section of the function is unchanged from our previous lab projects. It sets up a loop to iterate though every object in the scene's CObject array and fetches the CActor and CTriMesh pointers into

local variables. If the actor pointer is valid, we attach the object's controller to the actor and call the actor's AdvanceTime method to advance the timeline of any animation it may be playing.

```
void CScene::AnimateObjects( CTimer & Timer )
{
    ULONG i;

    // Process each object for coll det
    for ( i = 0; i < m_nObjectCount; ++i )
    {
        CObject * pObject = m_pObject[i];
        if ( !pObject ) continue;

        // Get the pointers
        CActor   * pActor = pObject->m_pActor;
        CTriMesh * pMesh  = pObject->m_pMesh;
        if ( !pActor && !pMesh ) continue;

        // Update actor?
        if ( pActor )
        {
            if ( pObject->m_pAnimController )
                 pActor->AttachController( pObject->m_pAnimController,
                                           false,
                                            pObject->m_pActionStatus );

            // Advance time
            pActor->AdvanceTime( Timer.GetTimeElapsed(), false );

        } // End if actor
```

The next section of code is only executed is the current object being processed has been registered with the collision system or the spatial tree. If so, then we will set the actor's matrix and force its absolute matrices to be updated. This is important because we shall see later that this is used to calculate the world space bounding box for the actor which we will need in order to update its position in the tree.

```
        if ( pObject->m_nObjectSetIndex > -1 || pObject->m_nTreeObjectIndex > -1 )
        {
            // Set world matrix and update combined frame matrices.
            if ( pActor ) pActor->SetWorldMatrix( &pObject->m_mtxWorld, true );
```

If the CObject has a valid collision system handle, we call the collision system's ObjectSetUpdated method to allow it to recalculate information about its dynamic object array.

```
            // Notify the collision system that this set of dynamic objects
            // positions, orientations or scale have been updated.
            if ( pObject->m_nObjectSetIndex > -1 )
                 m_Collision.ObjectSetUpdated( pObject->m_nObjectSetIndex );
```

The next bit is new. If the application is using a spatial tree and the current CObject being processed has a valid spatial tree handle stored in its m_nTreeObjectIndex structure, it means that this object has been

registered with the tree and therefore (as its position may have been updated by the AdvanceTime call) we should inform the spatial tree to recalculate the leaves in which this object is situated.

To instruct the spatial tree that the position of our object might have changed, we have to send the object's spatial ID and its world space bounding box into the ISpatialTree::UpdateTreeObject method. Until now, we have had no methods exposed from either CTriMesh or CActor that return the world space bounding box; we will add them to these classes at the end of this lesson. For now, let us just assume that both CTriMesh and CActor have a method called GetBoundingBox which is passed two vectors that will be filled with the world space extents of the object's bounding box.

```
            // Update the tree information
            if ( m_pSpatialTree && pObject->m_nTreeObjectIndex > -1 )
            {
                D3DXVECTOR3 vecMin, vecMax;

                // Retrieve the bounding box
                if ( pActor )
                    pActor->GetBoundingBox( vecMin, vecMax );
                else
                    pMesh->GetBoundingBox( vecMin, vecMax, &pObject->m_mtxWorld );
```

The CActor::GetBoundingBox method takes only two output parameters for the purposes of retrieving the world space box extents that encompass the entire actor hierarchy. The CTriMesh object (which may alternatively be stored in a CObject) has no way of returning a world space bounding box unassisted. It is typically an object space mesh and has no knowledge of its world space position. Therefore, the CTriMesh object will store a model space bounding box which can be converted into world space by passing a transformation matrix as the third parameter to its GetBoundingBox function. This function will simply use our handy TransformAABB method to convert the mesh's model space AABB into a world space AABB. The extents will then be returned using the first two output parameters.

At this point, whether the CObject contains a mesh or an actor, we have a world space bounding box which we can pass into the tree.

```
                // Update the spatial tree object
                m_pSpatialTree->UpdateTreeObject( pObject->m_nTreeObjectIndex,
                                                  vecMin,
                                                  vecMax );

            } // End if has object index

        } // End if actor exists

    } // Next Object
}
```

As you can see, we pass the UpdateTreeObject method the bounding box of the object and its spatial ID, which was issued at the time of registration. Later in the lesson we will see exactly how the bounding boxes are calculated for meshes and actors and how all these new spatial tree methods work.

## 15.6.5 Rendering Dynamic Objects

In the following code we see a section of the CScene::Render method. Since this is a large method we will only show the new additions which pertain to the rendering of the scene's objects. Remember that they have all been registered with the spatial tree as dynamic objects,

As we discussed in the previous section, situated near the top of the CScene::Render function, the ISpatialTree::ProcessVisibility function is called to traverse the tree and set the visible leaves based on the input camera's frustum. Earlier in the lesson we also learned that this function also builds the render batches for the static data stored in each leaf bin. We will see in a moment that this method can be updated to handle dynamic objects as well. When any leaf is found to be visible, any TreeObject structures stored in that leaf will have their Boolean pointers set to true. We know that the Boolean pointer in each TreeObject structure is actually a pointer to the CObject::m_bVisible member variable, so when the ProcessVisibility method returns, any CObjects that are in visible leaves will have their m_bVisible members set to true.

In the next section of code we show the call to the ProcessVisibility method followed by the loop that iterates through the CScene's CObject array and renders any that have had their visibility Booleans set to true by the spatial tree.

```
    // Update the leaf visibility  (sets CObject Booleans)
    m_pSpatailTree->ProcessVisibility( pCamera);

     … SNIP : Render spatial tree's static data here

    // Process each object
    for ( i = 0; i < m_nObjectCount; ++i )
    {
        CObject  * pObject = m_pObject[i];
        if ( !pObject ) continue;

        // Skip if this object is listed as not visible
        if ( !pObject->m_bVisible ) continue;

        // Flag this object as *not* visible for the next call to ProcessVisibility
        // but only if it has been registered with the tree
        if ( pObject->m_nTreeObjectIndex >= 0 ) pObject->m_bVisible = false;

        // Retrieve actor and mesh pointers
        CActor   * pActor = pObject->m_pActor;
        CTriMesh * pMesh  = pObject->m_pMesh;
        if ( !pMesh && !pActor ) continue;

        … SNIP : Set Object matrix
        …
        … For Each  Subset
        …
        …    pActor/pMesh -> DrawSubset ( … )
        …
        … End For Each Subset
```

```
    }

    // SNIP : Render any terrains here
```

The important point to notice in the above code snippet is that once we find a CObject that is in a visible leaf, we reset its visibility Boolean back to false prior to rendering it. This is so the Boolean will be returned to its default state when the ISpatialTree::ProcessVisibility method is called during the next frame render.

Now that we have had a high level look at how the spatial tree's dynamic object system will work, it should be clear that, from the application's perspective, things could not be easier. The application really only ever has to call two methods for each dynamic object. The InsertTreeObject method is called once (at load time in our example) to register the object with the tree, and the UpdateTreeObject method is called to update the position of the object inside the spatial tree when required. Let us now have a look at the changes we have had to make to both the ISpatialTree and ILeaf interfaces in order to add support for these concepts.

## 15.6.6 ILeaf - Adding Dynamic Object Support

The ILeaf abstract base class from which CBaseLeaf is derived only requires one extra method to be implemented in the derived classes so that application can query the list of TreeObjects for a given leaf. This method is called GetTreeObjectList and is highlighted in bold below.

***Excerpt from ISpatialTree.h***

```
class ILeaf
{
public:

    // Typedefs, Structures and Enumerators.
    typedef std::list<TreeObject*> TreeObjectList;


    // Constructors & Destructors for This Class.
    virtual ~ILeaf() {};     // forces derived classes to have virtual destructors


    // Public Pure Virtual Functions for This Class.
    virtual bool            IsVisible           ( ) const = 0;
    virtual unsigned long   GetPolygonCount     ( ) const = 0;
    virtual CPolygon *      GetPolygon          ( unsigned long nIndex ) = 0;
    virtual unsigned long   GetDetailAreaCount  ( ) const = 0;
    virtual TreeDetailArea *GetDetailArea       ( unsigned long nIndex ) = 0;

    virtual TreeObjectList& GetTreeObjectList   ( ) = 0;

    virtual void            GetBoundingBox      ( D3DXVECTOR3 & Min,
                                                  D3DXVECTOR3 & Max ) const = 0;

};
```

Notice that this method accepts no parameters and will return a list of all the TreeObject structures stored in this leaf. The return type TreeObjectList is a type definition for an STL list which stores TreeObject structure pointers. You can see this typedef at the top of the code shown above.

As you can see, from the application's perspective (which always works with the ISpatialTree interface), only one method has been added that allows it to retrieve the entire list of dynamic objects stored in that leaf. An application might, for example, employ a totally different rendering strategy than the one we have chosen. It might prefer to fetch the list of visible leaves from the tree and then loop through each of them individually. For each leaf, it can use this new method (GetTreeObjectList) to get the list of dynamic objects stored in that leaf and then fetch the context pointer from each TreeObject structure for further processing. This is an alternative to the Boolean pointer method we have chosen to use in Lab Project 14.1.

## 15.6.7 ISpatialTree - Adding Dynamic Object Support

Four new methods have been added to the ISpatialTree interface to allow the application to add, insert, and remove dynamic objects from the tree. There is also a method that allows the application to retrieve a list of all the leaves a given object is currently contained in.

We will not show the entire ISpatialTree class here since we have only added four methods. These are shown below.

***Excerpt from ISpatialTree.h***

```
virtual long            InsertTreeObject    ( TreeObject & Object ) = 0;

virtual void            UpdateTreeObject    ( long nObjectIndex,
                                              const D3DXVECTOR3 &BoundsMin,
                                              const D3DXVECTOR3 &BoundsMax)= 0;

virtual void            RemoveTreeObject    ( long nObjectIndex ) = 0;

virtual bool            GetTreeObjectLeaves ( long nObjectIndex,
                                              LeafList & List ) = 0;
```

These are all pure virtual methods which serve to define the functionality that must be implemented in the derived classes.

## 15.6.8 CBaseTree/CBaseLeaf - Adding Dynamic Object Support

The full implementation for dynamic object support will be contained within CBaseTree and CBaseLeaf. The derived classes will not require any additional code. We will start by examining the members and methods that will need to be added to CBaseLeaf first.

## 15.6.9 CBaseLeaf – The Source Code

CBaseLeaf will need three new member functions and a single member variable. In the following listing we do not show the entire declaration of CBaseLeaf, only the new members.

```
public:

    // Public Virtual Functions for This Class (from base).
    virtual TreeObjectList& GetTreeObjectList   ( );


    // Public Functions for This Class.
    void                    InsertTreeObject    ( TreeObject * pObject );
    void                    RemoveTreeObject    ( TreeObject * pObject );

protected:

    // Protected Variables for This Class
    TreeObjectList      m_TreeObjects;      // List of objects
```

**m_TreeObjects**
This member will be used to store a list of TreeObject structures. This list represents all the tree objects that are currently considered to be in this leaf by the CBaseTree. A dynamic object (a TreeObject) will be assigned to a leaf (or multiple leaves) when the application issues a call to the CBaseTree::UpdateTreeObject method. This method will traverse the tree with an input world space bounding box and collect a list of all leaves intersecting that box. The CBaseLeaf::InsertTreeObject method will then be called for each leaf in this list so that the tree object is added to each leaf's tree object list.

## InsertTreeObject – CBaseLeaf

This method will never be called by the application. It is used during a tree object update by CBaseTree when it determines that the tree object should be added to a leaf. The tree will issue a call to this method for each leaf in which the tree object is contained. This method is a simple function that just adds the passed TreeObject pointer to the leaf's internal list of tree objects.

```
void CBaseLeaf::InsertTreeObject( TreeObject * pObject )
{
    // Push this context pointer onto the end of the list
    m_TreeObjects.push_back( pObject );
}
```

## RemoveTreeObject – CBaseLeaf

This is another method that will never be called by the application, but is called by the tree during the update of a tree object. When the application issues a call to the CBaseTree::UpdateTreeObject method, the first thing this method will do is remove the tree object from any leaves to which it is currently assigned. We will see in a moment that, inside CBaseTree, each tree object is stored along with a list of leaves in which it is currently contained. Therefore, when its position needs to be updated, this leaf list is traversed and the CBaseLeaf::RemoveTreeObject method will be called to unhook the tree object from all its current leaves. The update method will then pass the bounding box of the tree object (in its new world space position) down the tree and collect a new list of leaves in which the object is now considered to be contained. This new leaf list is then traversed and the CBaseTree::InsertTreeObject method called for each. This will add the tree object to each of the new leaves as discussed above. Therefore, updating a tree object will essentially involve removing it from all its current leaves, finding a new list of leaves its bounding box intersects, and then adding the object to this new list of leaves. The code to CBaseLeaf::RemoveTreeObject, which is used in this process, is shown below.

```
void CBaseLeaf::RemoveTreeObject( TreeObject * pObject )
{
    // Remove this context pointer from the list
    m_TreeObjects.remove( pObject );
}
```

As you can see, it simply removes the passed TreeObject pointer from the leaf's tree object list. At that point, the tree object will no longer be considered to be in that leaf.

## GetTreeObjectList - CBaseLeaf

This method is required by the base class (ILeaf) and allows the application to retrieve the list of tree objects currently contained in the leaf. This method just returns the leaf's m_TreeObjects list.

```
ILeaf::TreeObjectList& CBaseLeaf::GetTreeObjectList( )
{
    // Just return the list.
    return m_TreeObjects;
}
```

As mentioned, this might prove useful if the application chooses a different rendering strategy than the one we are using.

## SetVisible – CBaseLeaf

We have examined this function a few times in this lesson in one form or another. You will recall that it is called by a derived class's UpdateTreeVisibility method every time a visible leaf is found during the visibility traversal. It is called to flag leaves as either invisible or visible.

As we know, this function sets the visible status of the leaf. If the leaf is visible, its triangle runs are added to the relevant leaf bins. After that, the leaf then adds its 'this' pointer to tree's visible leaf list. This is all unchanged from the previous version. All we have added to the bottom of the function is a couple of lines that loop through the list of tree objects stored in this leaf and sets their registered Booleans to true. The new lines of code are highlighted in bold at the bottom of the function listing.

```
void CBaseLeaf::SetVisible( bool bVisible )
{
    ULONG                 i, j;
    RenderData::Element * pElement;
    CLeafBin            * pLeafBin;
    RenderData          * pData;

    // Flag this as visible
    m_bVisible = bVisible;

    // If we're being marked as visible, inform the renderer
    if ( m_bVisible && m_nRenderDataCount > 0 )
    {
        // Loop through each renderable set in this leaf.
        for ( i = 0; i < m_nRenderDataCount; ++i )
        {
            pData    = &m_pRenderData[i];
            pLeafBin = pData->pLeafBin;

            // Loop through each element to render
            for ( j = 0; j < pData->ElementCount; ++j )
            {
                pElement = &pData->pElements[j];
                if ( pElement->PrimitiveCount == 0 ) continue;

                // Add this to the leaf bin
                pLeafBin->AddVisibleData( pElement->VBIndex,
                                          pElement->IndexStart,
                                          pElement->PrimitiveCount );
            } // Next Element
        } // Next RenderData Item
    } //  End if visible


    // Update tree object's if we're visible
    if ( m_bVisible )
    {
        // // Add this leaf to the tree's visible leaf list
        m_pTree->AddVisibleLeaf( this );
```

```
        TreeObjectList::iterator Iterator = m_TreeObjects.begin();
        for ( ; Iterator != m_TreeObjects.end(); ++Iterator )
        {
            TreeObject * pObject = *Iterator;
            if ( pObject->pbVisible ) *pObject->pbVisible = true;

        } // Next tree object

    } // End if we are visible
}
```

As you can see, we iterate through the TreeObject list stored in this leaf only if the leaf is visible. For each tree object, if it has a non-NULL Boolean pointer (pbVisible), then it means that the application would like this Boolean set to true when the object is visible. As we saw earlier, in our application, the Boolean pointer in each tree object will actually point to a CObject's visibility Boolean. This means that although the tree itself has no knowledge that this tree object actually represents a CObject structure, it still has the ability to set its visibility status to true so that the application knows it has to render this CObject during the CScene::Render function.

We have now looked at the minor changes to CBaseLeaf that provide support for the containment of dynamic objects. Next we will discuss the changes to the CBaseTree class where most of our dynamic object support functionality will be contained.

## 15.6.10 CBaseTree – The Source Code

We have seen that our leaves will maintain a list of TreeObject structures, so every leaf will know which dynamic objects it contains. However, CBaseTree will also store an array (STL vector) of all TreeObjects currently registered with the system, along with a list of leaves that object is currently contained within. This means that each leaf will have immediate access to the objects it contains, and each object will have immediate access to the list of leaves in which it is contained. This allows us to very efficiently return a list of leaves when the application issues a call to the ISpatialTree::GetTreeObjectLeaves method. This method is passed the ID of a tree object and returns its leaf list. The leaf list for each tree object is updated during the call to the CBaseTree::UpdateObject method, which we will see the code for in a moment.

## 15.6.11 The TreeObjectData Structure

A new structure will be needed so that CBaseTree can maintain a list of TreeObject structures paired with the object leaf lists. This new structure is defined in the CBaseTree namespace but is shown below on its own. It is called TreeObjectData and there will be an array of these structures stored in CBaseTree. Each element in this array will store the information for a tree object that is currently registered with the system.

***Excerpt from CBaseTree.h***

```
struct TreeObjectData             // Stores the data relating to a tree object
{
        TreeObject   Object;      // The 'context' for the tree object
        LeafList     Leaves;      // List of leaves in which the object exists
        bool         bInUse;      // This element is currently in use?
}
```

**TreeObject Object**

This member stores the TreeObject structure itself. This is the TreeObject that this TreeObjectData structure represents. Recall that the TreeObject structure has three members: a spatial ID that identifies the object to the system, a Boolean pointer that will be set to true when the object is visible, and a context pointer that can be used by the application to store any arbitrary data.

**LeafList Leaves**

This is a list of leaves in which the above TreeObject is currently considered contained. This leaf list is used to update the position of a tree object very efficiently. Because we store the list of leaves the object is currently contained in, when we wish to update the position of a tree object (which first involves removing it from all current leaves), we can iterate through this list calling the CBaseLeaf::RemoveTreeObject method before emptying the list. This will allow us to quickly empty this list and remove the object from all the leaves prior to an update. Then, the CBaseTree::UpdateTreeObject method will send the AABB of the tree object down the tree to build a new leaf list which is then stored in this member. We can then iterate through this list and call the CBaseLeaf::InsertTreeObject method for each leaf to add the TreeObject to each new leaf in which it is now contained.

Of course, the other useful thing about this member, beyond speeding up the tree object updates, is that it allows the application to retrieve a list of leaves for a given tree object. As this information is already compiled for every tree object, when the application requests the leaf list for a tree object by passing its spatial ID, the CBaseTree::GetTreeObjectLeaves method can simply search the TreeObjectData array for a TreeObject data structure with the matching ID and, if found, return the leaf list stored there.

**bInUse**

This is a Boolean that will tell the tree whether this particular TreeObjectData structure currently contains a tree object or whether it is empty and can be reused. This minimizes array resizing every time a tree object is removed. When a tree object is removed from the system (at the application's request), we do not delete its TreeObjectData structure and resize the array; we simply set this Boolean to false so that we know the next time we wish to add a new object, this TreeObjectData element can be used and its current data overwritten.

Below we see the new members and methods in CBaseTree that have been added to manage dynamic objects. As you can see, it implements the four methods required by ISpatialTree to allow the application to add, remove, and update dynamic objects within the tree. It contains only one new member, which is an array of TreeObjectData structures.

***Excerpt from CBaseTree.h***

```
public:

    virtual long            InsertTreeObject    ( TreeObject & Object );
    virtual void            UpdateTreeObject    ( long nObjectIndex,
                                                  const D3DXVECTOR3 & BoundsMin,
                                                  const D3DXVECTOR3 & BoundsMax );

    virtual void            RemoveTreeObject    ( long nObjectIndex );

    virtual bool            GetTreeObjectLeaves ( long nObjectIndex,
                                                  LeafList & List );

protected:

    // STL Typedefs
    typedef std::vector<TreeObjectData>     TreeObjectVector;

    // Protected Variables for This Class.
    TreeObjectVector            m_TreeObjects;
};
```

**TreeObjectVector m_TreeObjects**
This is an STL vector that stores TreeObjectData structures. Therefore, this array is basically used to store all of the tree's currently registered dynamic objects.

## InsertTreeObject – CBaseTree

This is the method that the application uses to register a dynamic object with the system. The caller fills out a TreeObject structure and passes it in and the function will look for a place to store it in the TreeObjectData array. It first searches though the m_TreeObjects vector to see if there are any TreeObjectData structures that are not currently in use by the system. If one if found, we break from the loop so that the loop variable *i* contains the index of this element.

```
long CBaseTree::InsertTreeObject( TreeObject & Object )
{
    ULONG i;

    // Loop through the vector and determine if there is a free slot
    for ( i = 0; i < m_TreeObjects.size(); ++i )
    {
        // Is this in use?
        if ( m_TreeObjects[i].bInUse == false ) break;

    } // Next Tree Object
```

If we could not find a free slot, then the loop would have run to completion. Either way, loop variable *i* will now contain the new index for where we wish to store the passed tree object. We store this index in the object's nTreeObjectIndex member so that on function return the application will have access to this

ID and can store it. This is a unique ID for this object within the system and the handle the application will use to refer to it when it wants to update its position within the tree.

```
    // Update the object's index to the new slot
    Object.nTreeObjectIndex = i;
```

If *i* equals the current size of the array, then it means that there was no free slot that can be reused, and we will need to instantiate a new TreeObjectData structure, store the passed TreeObject in it, and set its bInUse Boolean to true. We then add it to the end of the array.

```
    // Did we reach the end?
    if ( i == m_TreeObjects.size() )
    {
        TreeObjectData Data;

        // Populate a new data element
        Data.bInUse   = true;
        Data.Object   = Object;

        // Add a new element to the array
        m_TreeObjects.push_back( Data );

    } // End if no free slot
```

If we did find a free slot in the array, we will reuse it by storing the passed TreeObject in it and setting its bInUse Boolean to true. Notice in the following code that because this TreeObjectData structure was used previously by another object that has since be un-registered, the leaf list stored there will need to be emptied as it will not be valid for our new object.

```
    else
    {
        // Just re-use the free slot
        m_TreeObjects[i].bInUse                 = true;
        m_TreeObjects[i].Object                 = Object;
        m_TreeObjects[i].Leaves.clear();


    } // End if free slot found

    // Return the new object's index
    return i;
}
```

Finally, the function returns the ID that was assigned to the new tree object. This ID is just the position of the TreeObjectData structure in the tree's object list.

What is important to remember is that although this function is used to add a new dynamic object to the tree, when the function returns it will have not yet be assigned to any leaves. That is what the UpdateTreeObject method does, which we will examine next.

## UpdateTreeObject – CBaseTree

We saw earlier that this function is called by the application in the CScene::AnimateObjects method. It should be called whenever the world matrix of an object has been changed, or in the case of an animated actor, whenever its animation has been advanced. In the case of an animated actor, even if the application never changes its position in the scene, the individual meshes contained in its hierarchy may move or rotate. This would ultimately change the size of the actor's bounding box (the box that bounds all meshes in the actor) and if the bounding box has gotten bigger or smaller with respect to the previous frame update, the actor may now exist in more or less leaves than before.

The function is actually quite straightforward even though it would seem to have a difficult task to perform. It is passed the ID identifying the object that needs its leaves recalculated, and the world space bounding box of that object. The function uses the passed ID to fetch a pointer to the relevant TreeObjectData structure to update.

```
void CBaseTree::UpdateTreeObject( long nObjectIndex,
                                  const D3DXVECTOR3 & BoundsMin,
                                  const D3DXVECTOR3 & BoundsMax )
{
    LeafList::iterator Iterator;
    TreeObjectData *   pData = NULL;

    // Valid the specified data item.
    if ( nObjectIndex < 0 || nObjectIndex >= (signed)m_TreeObjects.size() ) return;
    if ( !m_TreeObjects[ nObjectIndex ].bInUse ) return;

    // Store pointer to object to save lookups
    pData = &m_TreeObjects[ nObjectIndex ];
```

As the object may have moved, we will first remove it from all leaves in which it is currently stored. As each TreeObjectData structure contains the TreeObject and its current leaf list, we just have to loop through that leaf list calling the CBaseLeaf::RemoveTreeObject method for each.

```
    // Loop through the leaves and remove this from the object list
    for ( Iterator = pData->Leaves.begin();
          Iterator != pData->Leaves.end(); ++Iterator )
    {
        CBaseLeaf * pLeaf = (CBaseLeaf*)(*Iterator);
        if ( !pLeaf ) continue;

        // Request that this object is removed from the leaf
        pLeaf->RemoveTreeObject( &pData->Object );

    } // Next Leaf

    // Clear the list
    pData->Leaves.clear();
```

We saw the code to the CBaseLeaf::RemoveTreeObject function a moment ago. It simply removes the passed TreeObject pointer from its list. After this loop ends, we will have visited every leaf that currently contains the object and removed its pointer from those leaves. After that, we empty the leaf list in the TreeObject as well. At this point our object has the same status as a newly registered object; it is not stored in any leaves and it does not have any leaves in its leaf list.

Now it is time to rebuild this information. We start by passing the TreeObject's empty leaf list into the CollectLeavesAABB method, along with the world space bounding box of the object. When this function returns, the object will have an updated leaf list.

```
    // Collect the new leaves
    CollectLeavesAABB( pData->Leaves, BoundsMin, BoundsMax );
```

Of course, our job is not quite done. Although the object now has an updated list of leaves, the leaves in this list still do not know that they contain the object. Therefore, we will loop through the new leaf list and call the CBaseLeaf::InsertTreeObject method to add the object to the object lists for each leaf in which it is currently contained.

```
    // Loop through the leaves and add this back to the newly discovered leaves
    for ( Iterator = pData->Leaves.begin();
          Iterator != pData->Leaves.end(); ++Iterator )
    {
        CBaseLeaf * pLeaf = (CBaseLeaf*)(*Iterator);
        if ( !pLeaf ) continue;

        // Request that this object is removed from the leaf
        pLeaf->InsertTreeObject( &pData->Object );

    } // Next Leaf
}
```

## GetTreeObjectLeaves - CBaseTree

This function can be used by the application to fetch the list of leaves (both visible and invisible) in which a tree object is currently contained. The application passes in the ID of the tree object it would like to retrieve a leaf list for and a leaf list to store the results.

```
bool CBaseTree::GetTreeObjectLeaves( long nObjectIndex, LeafList & List )
{
    // Valid the specified data item.
    if ( nObjectIndex < 0 || nObjectIndex >= (signed)m_TreeObjects.size() )
         return false;

    if ( !m_TreeObjects[ nObjectIndex ].bInUse ) return false;

    // Populate the list
    List = m_TreeObjects[ nObjectIndex ].Leaves;
```

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

This method uses the passed ID to fetch the TreeObjectData structure from the array and, provided this element is in use (i.e., it is a valid object), it returns the TreeObjectData's leaf list.

## RemoveTreeObject - CBaseTree

This method can be called by the application to remove a dynamic object from the tree. It is the mirror function to the InsertTreeObject method. Its single parameter is the ID of the tree object you would like to unregister from the system.

The first thing the method does is use the passed ID to fetch the corresponding TreeObjectData structure from the tree's object data array.

```
void CBaseTree::RemoveTreeObject( long nObjectIndex )
{
    LeafList::iterator Iterator;
    TreeObjectData *   pData = NULL;

    // Valid index?
    if ( nObjectIndex < 0 || nObjectIndex >= (signed)m_TreeObjects.size() ) return;

    // Store pointer to object to save lookups
    pData = &m_TreeObjects[ nObjectIndex ];
```

Now we loop through this tree object's leaf list and call the CBaseLeaf::RemoveTreeObject method for each one. This removes it from the object list in each leaf currently containing the object.

```
    // Remove it from any leaves it currently exists in
    for ( Iterator = pData->Leaves.begin();
          Iterator != pData->Leaves.end(); ++Iterator )
    {
        CBaseLeaf * pLeaf = (CBaseLeaf*)(*Iterator);
        if ( !pLeaf ) continue;

        // Request that this object is removed from the leaf
        pLeaf->RemoveTreeObject( &pData->Object );

    } // Next Leaf
```

Finally, we set this TreeObjectData structure's bInUse Boolean to false and empty its leaf list.

```
    // Just re-use the free slot next time someone inserts
    pData->bInUse = false;
    pData->Leaves.clear();
}
```

We have now covered all the new methods and code that are involved in adding dynamic object support to CBaseTree.

## 15.6.12 Conclusion – Dynamic Objects

After an admittedly arduous journey, we have completed the code for our ISpatialTree derived classes. Along the way we have seen that CBaseTree provides nearly all of the core functionality for any tree type we might care to derive from it in the future.

One interesting item we encountered in this last section was that CBaseTree views dynamic objects in a very abstract way. It has no idea what the TreeObject structure represents and does not need to. We do not even use the context pointer of this structure in our lab project, only its Boolean pointer. Therefore, we have discovered that a dynamic object used in this way is really just a Boolean pointer with an assigned ID. It is ultimately attached to some number of leaves every time the UpdateTreeObject method is called, but it remains quite a separate concept. The TreeObject does not even store a bounding volume for the object it represents, as one might suspect. Instead, this bounding volume is passed into the UpdateTreeObject method by the application whenever it wishes to inform the tree that the object has moved and its Boolean pointer should now be assigned to a new set of leaves.

Of course, the dynamic object system we have developed does place a particular burden on the application. In order for the application to update the position of a CObject, it must have access to the world space bounding box of the object it contains (either a CTriMesh or a CActor). Up until this point in the course however, CTriMesh and CActor have never exposed a method for retrieving any world space bounding box, so we will need to add these now before concluding the lesson.

# 15.7 CTriMesh Revisited – World Space Bounding Boxes

CTriMesh will now have two new members added to it. These will be 3D vectors that describe the *object space* AABB of the mesh. As a mesh is by its very nature an object space concept, there is no way it can know about its world space position or size and therefore, the object space box will be created and stored instead. CTriMesh will then expose a method called GetBoundingBox which allows for a world transformation matrix to be passed. This function will transform the object space bounding box into world space before returning it to the application. We saw CTriMesh::GetBoundingBox being used earlier when we examined the changes to the CScene::AnimateObjects method. This method would fetch the world space bounding box from the CTriMesh (if one was stored in the current CObject structure being updated) and pass it into the ISpatialTree::UpdateTreeObject. When the CTriMesh::GetBoundingBox method was called, it was passed the world matrix of the owner object. This meant we would get back the world space AABB for the mesh.

Here are the new members and methods added to CTriMesh.

```
protected:
D3DXVECTOR3         m_vecBoundsMin;
D3DXVECTOR3         m_vecBoundsMax;

public:

HRESULT UpdateBoundingBox();

void    GetBoundingBox(D3DXVECTOR3 &BoundsMin,
                       D3DXVECTOR3 &BoundsMax,
                       D3DXMATRIX * pMatrix = NULL ) const;
```

## UpdateBoundingBox - CTriMesh

Our application will never call this function (although it can if it would like to rebuild the object space bounding box of the mesh for some reason) as it is called automatically by the CTriMesh::LoadMeshFromX function just before returning. The function is like many we have seen before. The first thing it does is query the D3DXMesh managed by the CTriMesh object for an ID3DXBaseMesh interface. This allows us to work with both regular and progressive meshes using the base interface.

```
HRESULT CTriMesh::UpdateBoundingBox( )
{
    HRESULT           hRet;
    UCHAR           * pVertices  = NULL;
    ULONG             nVertexStride, nVertexCount;
    D3DXVECTOR3       vecPos;
    ULONG             i;

    // Retrieve the mesh
    LPD3DXBASEMESH pMesh = NULL;

    // What type of mesh?
    if ( m_pMesh )
    {
        // Query the interface to get back the base mesh.
        hRet = m_pMesh->QueryInterface( IID_ID3DXBaseMesh, (void**)&pMesh );
        if ( FAILED( hRet ) ) return hRet;

    } // End if standard mesh
    else if ( m_pPMesh )
    {
        // Query the interface to get back the progressive mesh
        hRet = m_pPMesh->QueryInterface( IID_ID3DXBaseMesh, (void**)&pMesh );
        if ( FAILED( hRet ) ) return hRet;

    } // End if progressive mesh
    else
    {
        // Just return
```

```
        return D3D_OK;

    } // End if no mesh
```

We initialize the mesh's bounding box to extreme starting values (an inside-out box).

```
    // Reset the bounding box
    m_vecBoundsMin = D3DXVECTOR3(  FLT_MAX,  FLT_MAX,  FLT_MAX );
    m_vecBoundsMax = D3DXVECTOR3( -FLT_MAX, -FLT_MAX, -FLT_MAX );
```

We then get the number of vertices and the number of bytes each vertex consumes in memory before locking the mesh's vertex buffer to get a pointer to its vertex data.

```
    // Retrieve additional info we need for processing
    nVertexStride = pMesh->GetNumBytesPerVertex( );
    nVertexCount  = pMesh->GetNumVertices( );

    // Lock the vertex buffer
    hRet = pMesh->LockVertexBuffer( D3DLOCK_READONLY, (void**)&pVertices );
    if ( FAILED(hRet) ) { pMesh->Release(); return hRet; }
```

Now we will iterate through every vertex in the vertex buffer and copy the positional data into a D3DXVECTOR3 called vecPos.

```
    // Compute this frame's bounding box
    for ( i = 0; i < nVertexCount; ++i, pVertices += nVertexStride )
    {
        // Retrieve the position of this vertex in it's reference pose
        vecPos = *(D3DXVECTOR3*)pVertices;
```

We then test the components of this vertex against the currently recorded maximum and minimum box extents and adjust them accordingly.

```
        // Test it against the frame bounding box and update if necessary
        if ( vecPos.x < m_vecBoundsMin.x ) m_vecBoundsMin.x = vecPos.x;
        if ( vecPos.y < m_vecBoundsMin.y ) m_vecBoundsMin.y = vecPos.y;
        if ( vecPos.z < m_vecBoundsMin.z ) m_vecBoundsMin.z = vecPos.z;
        if ( vecPos.x > m_vecBoundsMax.x ) m_vecBoundsMax.x = vecPos.x;
        if ( vecPos.y > m_vecBoundsMax.y ) m_vecBoundsMax.y = vecPos.y;
        if ( vecPos.z > m_vecBoundsMax.z ) m_vecBoundsMax.z = vecPos.z;

    } // Next Vertex
```

After this loop finishes, the bounding box will fit every vertex in the mesh and our job is complete. All we have to do now is unlock the vertex buffer and return.

```
    // Unlock the vertex buffer
    pMesh->UnlockVertexBuffer();

    // Release the pointer to the mesh we retrieved
```

```
    pMesh->Release();

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

It should be noted that while this method need never be called if you are populating the CTriMesh using its CTriMesh::LoadMeshFromX method (it calls this method automatically), if you are procedurally building a CTriMesh then you will want to call this method after you have added all the vertex data and built its underlying D3DXMesh. Another time you will want to call this method is if you detach the underlying D3DXMesh and attach a new one.

## GetBoundingBox - CTriMesh

Although the CTriMesh object stores its model space bounding box, we want a function that can optionally return the world space bounding box if needed. A CTriMesh object has no information about where the mesh lives in the world, so we will add a matrix pointer parameter to its GetBoundingBox function. This allows the application to pass a world matrix that will be used to transform the model space bounding box into world space before returning it to the caller. Luckily, we have already written the TransformAABB method that performs this very task and as such, the CTriMesh::GetBoundingBox uses it to optionally perform the world space transformation.

```
void CTriMesh::GetBoundingBox( D3DXVECTOR3 & BoundsMin,
                               D3DXVECTOR3 & BoundsMax,
                               D3DXMATRIX * pMatrix /* = NULL */ ) const
{
    BoundsMin = m_vecBoundsMin;
    BoundsMax = m_vecBoundsMax;

    // Transform if a matrix is provided
    if ( pMatrix ) MathUtility::TransformAABB( BoundsMin, BoundsMax, *pMatrix );
}
```

As you can see, the function is passed the 3D vectors that will receive the resulting AABB extents. First we copy the model space bounding box of the mesh into these vectors. If the caller passed a matrix, this bounding box is then transformed by that matrix prior to the function returning.

# 15.8 CActor Revisited – World Space Bounding Boxes

Adding a function to our actor that will return its world space bounding box will not be quite as simple as it was for our mesh. The bounding box it returns must be large enough to encompass all the meshes contained in its hierarchy. At first we might think that this could be done at actor creation time by traversing the tree to find all mesh containers and then building a box to contain the vertices in those meshes. That would certainly work if the actor was not animated, but as we know, actors can be

animated and this changes things. If we imagine an actor that has been created to store a hierarchy of meshes that all rotate and move, we can see that the size of the actor's bounding box would changed during animation updates. Imagine a skinned character for example. It would have one bounding box when its arms are by its side and a different one when it holds its arms out to each side. An even better example of the drastic size changes that can occur to an actor's bounding box would be an actor that models a space ship on a launching pad (like our lab project in Chapter 9). At the start of the animation, the craft would be on the launch pad with a small bounding box encompassing the launch pad and the spaceship. As the animation plays, the spaceship takes off and flies off into the distance. As both the launch pad and the actor are mesh containers within the same actor in this example, the bounding box of the actor would have to dynamically grow to contain the craft as it flies off into the distance.

Traversing the actor hierarchy whenever its animation is updated and calculating a new world space bounding box at the per-mesh level is out of the question if we want to do this quickly. Instead, we will use an approach that will take advantage of the fact that the actor has a spatial tree of its own that can be traversed and updated. Our design approach will be very familiar to you since it is identical in concept to the way we generate the world matrices (combined matrices) for each frame.

Each frame in the tree will now store two sets of bounding box extents (four vectors). Our new D3DXFRAME_MATRIX structure is shown below.

```
struct D3DXFRAME_MATRIX : public D3DXFRAME
{
    D3DXMATRIX   mtxCombined;    // Combined matrix for this frame.
    D3DXVECTOR3  vecBoundsMin;   // Bounding box (in world space)
    D3DXVECTOR3  vecBoundsMax;   // Bounding box (in world space)

    D3DXVECTOR3  vecObjectMin;   // Bounding box (in object space)
    D3DXVECTOR3  vecObjectMax;   // Bounding box (in object space)
    bool         bObjectBounds;  // This bounding box describes the extents
                                 // of a physical object?
};
```

## 15.8.1 The Model Space Bounding Boxes

vecObjectMin and vecObjectMax will be calculated automatically when the actor's data is first loaded from the X file (via a call from the LoadActorFromX method to the BuildBoundingBoxes method). It will store the model space bounding box of any frame that directly stores mesh data. But what does the bounding box of a frame represent? In our case it will represent a bounding box that is large enough to contain all the mesh containers ***directly*** attached to a given frame. Only frames in the hierarchy that have mesh containers attached will store a model space bounding box, so there will not yet be any parent/child relationship as we see in a spatial tree of bounding volumes.

During the building phase of these boxes, the tree will be traversed and any frame that has a mesh container attached will have a bounding box generated for that mesh (or list of meshes if multiple mesh containers are assigned to the same frame). That model space box with be stored in the owner frame. You will notice that our updated frame structure also stores a new Boolean called bObjectBounds. This

will be set to true only for frames that store a model space bounding box in the vecObjectMin and vecObjectMax members (i.e., frames that have mesh data directly attached to them). However, we must also remember that the actor may contain a skinned mesh, in which case things are a little different.

Up until now we have been talking about the actor and its model space bounding boxes in the simplest form. That is, any frame that has a mesh container attached will also store a model space bounding box describing the position and size of that mesh in the actor's reference pose. However, if we think about the bones of an actor, they do not have mesh containers attached to them (usually the entire skin is stored in the tree attached to an arbitrary frame; most often the root) so does that mean bones should not store model space bounding boxes either? It certainly does not!

Although a bone may not have a mesh container structure attached to it, it does represent some portion of a mesh. For example, imagine a bone that has all the vertices of a skinned character's elbow mapped to it. That bone should contain a bounding box that describes the position and size of the elbow in the default pose. Therefore, we have two cases where a frame will store a model space bounding box. The first is when the frame has a mesh container attached and the second is when the frame is being used as a bone by a skinned mesh. In the latter case, its model space bounding box will bound the section of vertices (in their model space reference pose) in the skin that it influences.

For a skinned character, we can easily imagine having bounding boxes surrounding all the bones in the reference pose (see Figure 15.8). The black box shows what we are trying to ultimately achieve -- a bounding box that encompasses the entire actor, in world space.

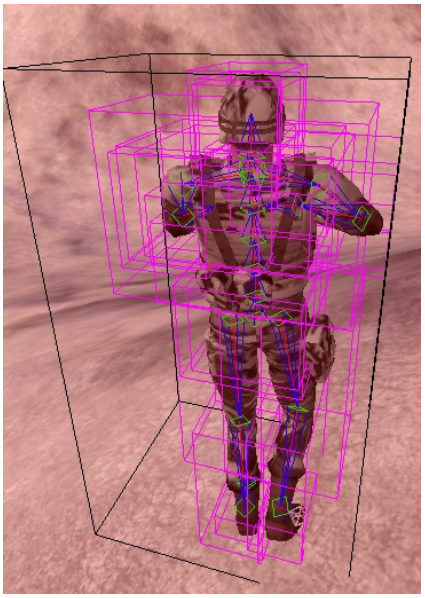


**Figure 15.8**

So we have determined that the object space bounding boxes of any frame that contains (or is attached to) vertex data will be calculated in a single function called once when the actor is first created. This function is called CActor::BuildBoundingBoxes and it is called from the bottom of the CActor::LoadActorFromX function.

The calculation of these boxes will not be fast enough for real-time work, which is why we are lucky it only has to be performed once. We will have to traverse the tree searching for frames that contain mesh containers and need to have an object space box created for them. When a mesh container is found, we then test to see if the mesh is a skin. If it is not a skin, our task is simple: we lock the mesh's vertex buffer and iterate through each vertex, adjusting the size of the frame's object space box extents until it is large enough to encompass all vertices of that mesh. If multiple mesh containers are attached to the frame, they must all be tested in this way and the bounding box of the frame will grow to fit all these attached meshes.

When the mesh container stores a skin, things are a little bit different. We have to loop through each bone and retrieve the vertices that are influenced by it. Once we have a list of vertices for the current bone we are processing we must transform those vertices into bone space. Remember, the skin itself will be in its own model space at this point and we want its vertices in the space of the actor. After

transforming the vertices attached to the current bone into bone space, we compute its bounding box and grow the extents of the frame's box if it is not large enough to contain it. We do this same step for each bone that influences the mesh.

We will now look at the code to the CActor::BuildBoundingBoxes method. One thing to bear in mind when you see this code is that it never initializes the bounding boxes of any frames to default values. This is done when the frame is first created in the CAllocateHierarchy::CreateFrame callback. This is quite important because a single bone may influence more than one skin. Resetting its bounding box before calculating the extents of any particular skin's vertices would cancel out the contribution of other skins.

## BuildBoundingBoxes - CActor

This method is a recursive function that is called from the CActor::LoadActorFromX function. It is passed a pointer to the root frame and then traverses the frame hierarchy looking for frames that contain meshes or that influence the vertices of a skinned mesh. Once found, it grows the model space bounding box of the owner frame to contain any meshes/vertices it directly influences. We will cover the code one section at a time.

The first thing we do is loop through the list of mesh containers stored at the frame. If there are no mesh containers attached to this frame, no action is taken and the frame does not have a model space bounding box calculated for it. At the end of the function, it simply steps into the sibling and child lists.

At the start of the mesh container traversal loop, we determine whether the mesh stored in this container is a progressive mesh or a regular mesh, so that we can access the correct pointer in the D3DXMESHDATA structure. We then query the mesh for a D3DXBASEMESH interface so that we can work with both mesh types through a single interface for the remainder of the function.

```
HRESULT CActor::BuildBoundingBoxes( LPD3DXFRAME pFrame )
{
    HRESULT             hRet;
    D3DXFRAME_MATRIX * pMtxFrame  = (D3DXFRAME_MATRIX*)pFrame;
    D3DXFRAME_MATRIX * pBoneFrame = NULL;
    UCHAR            * pVertices  = NULL;
    ULONG               nVertexStride, nVertexCount, nBoneCount;
    ULONG               nInfluenceCount, i, j;
    D3DXVECTOR3         vecPos;

    // If this has a mesh container , we'll check for skinning etc.
    D3DXMESHCONTAINER_DERIVED * pContainer =
                               (D3DXMESHCONTAINER_DERIVED*)pFrame->pMeshContainer;

    for ( ; pContainer;
            pContainer=(D3DXMESHCONTAINER_DERIVED*)pContainer->pNextMeshContainer )
    {
        // Retrieve the mesh
        LPD3DXBASEMESH pMesh = NULL;
```

```
        switch( pContainer->MeshData.Type )
        {
            case D3DXMESHTYPE_MESH:

                // Skip if no mesh stored here
                if ( !pContainer->MeshData.pMesh ) continue;

                // Query the interface to get back the base mesh.
                hRet =
                pContainer->MeshData.pMesh->QueryInterface( IID_ID3DXBaseMesh,
                                                            (void**)&pMesh );
                if ( FAILED( hRet ) ) return hRet;
                break;

            case D3DXMESHTYPE_PMESH:

                // Skip if no mesh stored here
                if ( !pContainer->MeshData.pPMesh ) continue;

                // Query the interface to get back the progressive mesh
                hRet =
                pContainer->MeshData.pPMesh->QueryInterface( IID_ID3DXBaseMesh,
                                                             (void**)&pMesh );
                if ( FAILED( hRet ) ) return hRet;
                break;

            default:

                // We don't support other types
                continue;

        } // End mesh type
```

At this point we have our mesh pointer, so we will retrieve the vertex count of its vertex buffer and the stride of each vertex before locking the vertex buffer.

```
        // Retrieve additional info we need for processing
        nVertexStride = pMesh->GetNumBytesPerVertex( );
        nVertexCount  = pMesh->GetNumVertices( );

        // Lock the vertex buffer
        hRet = pMesh->LockVertexBuffer( D3DLOCK_READONLY, (void**)&pVertices );
        if ( FAILED(hRet) ) { pMesh->Release(); return hRet; }
```

Now that we have a pointer to the vertices, we have to figure out if this is a normal mesh attached to this frame or if the mesh is a skin that may be influenced by many other bones in the hierarchy. If a skin is not stored here then we know that the container's pSkinInfo pointer will be NULL. In the code snippet we see the code that handles the non-skin case. It loops through every vertex in the vertex buffer and grows the frame's bounding box to encompass the vertices. As mentioned previously, notice that we do not initialize the frame's object space bounding box extents. This is done when the frame itself is first created in the CAllocateHierarchy::CreateFrame method (called by D3DX during the loading of the hierarchy). We must not do that here as this may be one of many meshes attached to this frame. As we

know, it is possible for a frame to have a whole list of mesh containers attached via their pNextMeshContainer pointers.

```
        // Skinned mesh?
        if ( !pContainer->pSkinInfo )
        {
            // Compute this frame's bounding box
            // Note: This frame may have more than one mesh container,
            // boxes initialized during the 'CAllocateHierarchy::CreateFrame' call.
            for ( i = 0; i < nVertexCount; ++i, pVertices += nVertexStride )
            {
                // Retrieve the position of this vertex in it's reference pose
                vecPos = *(D3DXVECTOR3*)pVertices;

                // Test it against the frame bounding box and update if necessary
                if ( vecPos.x < pMtxFrame->vecObjectMin.x )
                     pMtxFrame->vecObjectMin.x = vecPos.x;

                if ( vecPos.y < pMtxFrame->vecObjectMin.y )
                     pMtxFrame->vecObjectMin.y = vecPos.y;

                if ( vecPos.z < pMtxFrame->vecObjectMin.z )
                     pMtxFrame->vecObjectMin.z = vecPos.z;

                if ( vecPos.x > pMtxFrame->vecObjectMax.x )
                     pMtxFrame->vecObjectMax.x = vecPos.x;

                if ( vecPos.y > pMtxFrame->vecObjectMax.y )
                    pMtxFrame->vecObjectMax.y = vecPos.y;

                if ( vecPos.z > pMtxFrame->vecObjectMax.z )
                    pMtxFrame->vecObjectMax.z = vecPos.z;

            } // Next Vertex

            // Frame has applicable bounding box
            pMtxFrame->bObjectBounds = true;

        } // End if standard mesh
```

As you can see in the above code, after every vertex has been tested against the current extents of the frame's bounding box (and adjusted where necessary), we set the frame's bObjectBounds member to true. We will see later that this Boolean is what will tell our CActor::UpdateFrames method that the frame contains an object space bounding box which must be transformed into world space and propagated up the tree to the root node (more on this later).

The next section of code shows what happens when the mesh container stores a skin. In this case, we first use the ID3DXSkinInfo::GetNumBones method to retrieve the number of bones (frames) in the hierarchy that influence the vertices in this mesh. We then initiate a loop to step through each bone.

```
        else
        {
            LPD3DXSKININFO pSkinInfo = pContainer->pSkinInfo;
```

```
        // Loop through each bone in the skin info
        nBoneCount = pSkinInfo->GetNumBones();

        for ( i = 0; i < nBoneCount; ++i )
        {
```

Now that we know the index of the current bone we are processing in the ID3DXSkinInfo's array of bone information, we call its GetBoneName method to retrieve the name of the current frame we are processing. This is then passed into the CActor::GetFrameByName method which traverses the hierarchy searching for the frame and returns a pointer to that frame when found.

```
            // Attempt to retrieve the frame for this bone
            pBoneFrame=(D3DXFRAME_MATRIX*)
                        GetFrameByName(pSkinInfo->GetBoneName( i ) );

            if ( !pBoneFrame ) continue;

            // Retrieve the total number of vertices that this bone influences
            nInfluenceCount = pSkinInfo->GetNumBoneInfluences( i );

            if ( nInfluenceCount == 0 ) continue;
```

After we have a pointer to the current frame/bone we are processing, we pass its index into the ID3DXSkinInfo::GetNumBoneInfluences method. This will return the number of vertices in this skin that are influenced by that bone (which we store in nInfluenceCount). If the return value is zero, then this bone does not influence the skin in any way and we can skip to the next iteration of the loop and process the next bone.

Now that we know how many vertices in the skin are influenced by this bone, we need to fetch the indices of these vertices so that we can adjust the bone's object space bounding box. Therefore, we allocate an array of indices large enough to hold an index for each vertex influenced by the current bone and we allocate an array of floats that will be used to store the weight for each vertex. We pass pointers to these two arrays, along with the bone index, into the ID3DXSkinInfo::GetBoneInfluence method. This function will fill the passed arrays with the indices and weights of all vertices that are influenced by the current bone we are processing.

```
            // Allocate enough space for influences
            ULONG * pIndices = new ULONG[ nInfluenceCount ];
            if ( !pIndices )
            { pMesh->UnlockVertexBuffer();
              pMesh->Release();
              return E_OUTOFMEMORY;
            }

            float * pWeights = new float[ nInfluenceCount ];
            if ( !pWeights )
            {
                delete []pIndices;
                pMesh->UnlockVertexBuffer();
                pMesh->Release();
```

```
                    return E_OUTOFMEMORY; }

                // Retrieve the influence array
                hRet = pSkinInfo->GetBoneInfluence( i, pIndices, pWeights );
                if ( FAILED(hRet) )
                {
                    // Clean up and continue
                    delete []pIndices;
                    delete []pWeights;
                    pMesh->UnlockVertexBuffer();
                    pMesh->Release();
                    continue;

                } // End if failed to get influences
```

Now that we have the array of indices, we can loop through each one and fetch the position of the vertex at the corresponding index from the vertex buffer. We will store this in a temporary 3D vector, as shown below.

```
                // Loop through each influence
                for ( j = 0; j < nInfluenceCount; ++j )
                {
                    // Retrieve vertex in it's correct reference pose
                    vecPos = (D3DXVECTOR3&)(pVertices[pIndices[j]*nVertexStride]);
```

We now have the vertex position, but this in the local space of the skin model itself, not in the local space of the frame/bone to which it is attached. Therefore, we retrieve the bone offset matrix for the current bone and transform the vertex into bone space. We then test the position of the vertex against the object space (bone space) bounding box and grow its extents where necessary.

```
                    D3DXVec3TransformCoord( &vecPos,
                                            &vecPos,
                                            &pContainer->pBoneOffset[i] );

                    // Test against the frame bounding box and update if necessary
                    if ( vecPos.x < pBoneFrame->vecObjectMin.x )
                        pBoneFrame->vecObjectMin.x = vecPos.x;

                    if ( vecPos.y < pBoneFrame->vecObjectMin.y )
                        pBoneFrame->vecObjectMin.y = vecPos.y;

                    if ( vecPos.z < pBoneFrame->vecObjectMin.z )
                        pBoneFrame->vecObjectMin.z = vecPos.z;

                    if ( vecPos.x > pBoneFrame->vecObjectMax.x )
                        pBoneFrame->vecObjectMax.x = vecPos.x;

                    if ( vecPos.y > pBoneFrame->vecObjectMax.y )
                        pBoneFrame->vecObjectMax.y = vecPos.y;

                    if ( vecPos.z > pBoneFrame->vecObjectMax.z )
                        pBoneFrame->vecObjectMax.z = vecPos.z;
```

```
                } // Next Influence
```

After the index loop ends, we have correctly adjusted the object space bounding box for the current bone to contain all the vertices in the skin that are influenced by it. All we have to do now is set the bone's pObjectBounds Boolean to true and delete the temporary index and weight arrays.

```
                // Frame has applicable bounding box
                pBoneFrame->bObjectBounds = true;

                // Clean up memory
                delete []pIndices;
                delete []pWeights;

            } // Next Bone

        } // End if skinned mesh

        // Unlock the vertex buffer
        pMesh->UnlockVertexBuffer();

        // Release the pointer to the mesh we retrieved
        pMesh->Release();

    } // Next mesh container
```

After we have processed every bone that influences the skin and calculated the bounding box for each one, we unlock the vertex buffer and release the mesh interface we acquired.

Finally, the function continues its traversal of the tree looking for more mesh containers.

```
    // Has a sibling frame?
    if (pFrame->pFrameSibling != NULL)
    {
        hRet = BuildBoundingBoxes( pFrame->pFrameSibling );
        if ( FAILED(hRet) ) return hRet;

    } // End if has sibling

    // Has a child frame?
    if (pFrame->pFrameFirstChild != NULL)
    {
        hRet = BuildBoundingBoxes( pFrame->pFrameFirstChild );
        if ( FAILED(hRet) ) return hRet;

    } // End if has child

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

Any frames which do not have meshes attached to them or are not used to influence skins do not have an object space bounding box. Consequently, they will have their bObjectBounds Booleans set to false.

## 15.8.2 World Space Bounding Boxes

So far we have an object space bounding box stored at every frame that contains a mesh. The problem is that we need our actor to return a world space bounding box for the entire actor that will automatically have its size updated when the actor animates. We can see in Figure 15.9 that when the character is animated, the actor's bounding box changes in size to encompass the world space bounding boxes of all its child frames.

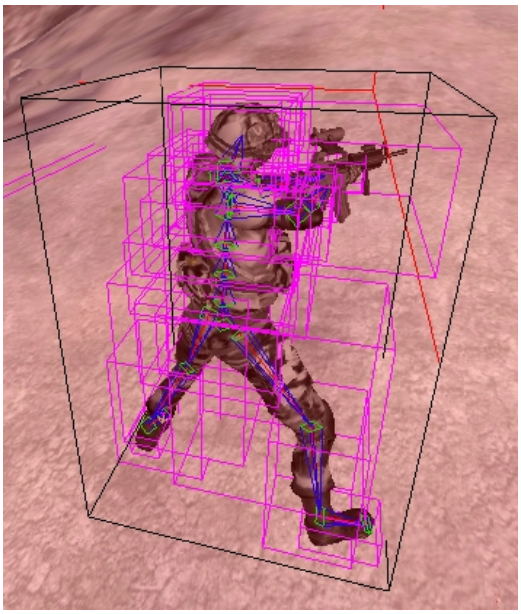


**Figure 15.9**

In Figure 15.9, the purple bounding boxes are *not* the bounding boxes we have just calculated; they are those bounding boxes after they have been transformed into world space. These world space bounding boxes (which need to be updated every time the actor is animated) are then merged together to compute the final world space bounding box for the entire actor.

The vecBoundsMax and vecBoundsMin members of our frame structure will be updated every time the actor is updated inside the CActor::UpdateFrameMatrices call. This is the method that traverses the hierarchy, combining matrices to ultimately store a world matrix at each frame. In many ways, our bounding boxes will work in exactly the same way as the relative and absolute matrices we are used to working with.

Recall that when the actor is first created, each frame contains a matrix that describes its position in parent relative space. However, each frame also stores another matrix that is updated every time the actor is moved. This update happens in the UpdateFrameMatrices method. You will recall that this function is passed a world matrix which is initially combined with the root frame's matrix to generate the world matrix of the root frame. This world matrix is then passed down to the children, where it is combined with their relative matrices to create the world matrices for each frame, and so on right down the tree. When UpdateFrameMatrices returns, every frame in the tree will store an absolute world matrix.

Our bounding boxes will basically work the same way. Our actor will store bounding box extents which will be used to store the bounding box of the entire actor. These will be updated every time the application calls the SetWorldMatrix method, which as we know, calls the UpdateFrameMatrices method. The UpdateFrameMatrices method will now have some code added so that as it steps through the tree, it will calculate the world space bounding box for every frame in the tree. The world space bounding box of a frame will be a box that has been adjusted to contain the world space bounding boxes of all frames beneath it in the hierarchy. When this process works its way back up the root, we will have a box that is in world space that encompasses every frame in the tree. The following description takes you through how it will work…

As we step into a frame we will calculate its absolute world matrix and store it in the frame as normal. We will then traverse into the sibling and child lists. When a frame is reached that has its bObjectBounds Boolean set to true, we will transform its object space bounding box into world space using our TransformAABB method. We will then store the world space bounding box in the frame and will pass it back to the parent frame when the function returns. The parent frame, having received the world space bounding box of its child(ren) will then adjust the returned box by transforming its own object space bounding box into world space (if it contains one) and grow the box to contain both its own box and the box returned by its child. The combined world space box will then be returned to its parent where the same happens again.

By positioning the bounding box propagation code after the lines that traverse into the child and sibling lists, we make sure that we are calculating the bounding box *not* on the way down the tree (as is the case with the frame matrices), but on our way back *up* the tree. As we begin unwinding the call stack and work our way back up the tree, we can combine the world space bounding box returned by the children with the current frame's own world space bounding box. Of course the world space bounding box returned by the children will also have been combined with the world space bounding boxes of all their children, and so on. What we end up with is a spatial hierarchy; a tree of bounding boxes such that every frame in the tree stores a world space bounding box that encompasses the world space bounding boxes of all the frames beneath it in the tree. The bounding box finally returned from the CActor::UpdateFrameMatrices method back to the CActor::SetWorldMatrix method is the world space bounding box of the entire actor in its current pose. We can then store this information away in the actor's bounding box extents member variables.

> **Note:** This bottom-up calculation approach is commonly used when assembling other types of popular bounding volume hierarchies like scene graphs, AABB trees, etc. We will examine some of these other hierarchy types in Module III of this series.

CActor has had two new members added to store the world space bounding box of the actor in its current pose. These two vectors will be populated by the data returned from the UpdateFrameMatrices call.

***Excerpt From CActor***

```
D3DXVECTOR3  m_vecBoundsMin;   // Minimum bounding box extents (in world space)
D3DXVECTOR3  m_vecBoundsMax;   // Maximum bounding box extents (in world space)

void   GetBoundingBox ( D3DXVECTOR3 & BoundsMin, D3DXVECTOR3 & BoundsMax ) const;
void   UpdateFrameMatrices(        LPD3DXFRAME   pFrame,
                           const   D3DXMATRIX  * pParentMatrix,
                                   D3DXVECTOR3 * pFrameMin /* = NULL */,
                                   D3DXVECTOR3 * pFrameMax /* = NULL */ )
```

CActor has also had two new method added, which we will discuss next.

## GetBoundingBox - CActor

The GetBoundingBox method returns the world space bounding box for the actor. We saw earlier that our CScene::AnimateObjects method called this function to send the box into the ISpatialTree:UpdateTreeObject method so that the leaf list for the object could be updated.

```
void CActor::GetBoundingBox( D3DXVECTOR3 &BoundsMin, D3DXVECTOR3 &BoundsMax ) const
{
    BoundsMin = m_vecBoundsMin;
    BoundsMax = m_vecBoundsMax;
}
```

This method will only be valid once the application has called CActor::SetWorldMatrix or one of the other CActor methods that automatically calls the UpdateFrameMatrices method. It will always contain the world space bounding box calculated during the last CActor::SetWorldMatrix call.

## SetWorldMatrix – CActor (Updated)

The application uses the CActor::SetWorldMatrix method to set the world matrix of the actor and optionally rebuild the world matrices of each frame in the hierarchy. This new version of the code initializes the actor's bounding box extents and then sends them into the UpdateFrameMatrices method.

```
void CActor::SetWorldMatrix( const D3DXMATRIX * mtxWorld /* = NULL */,
                             bool UpdateFrames /* = false */ )
{
    // Store the currently set world matrix
    if ( mtxWorld )
        m_mtxWorld = *mtxWorld;
    else
        D3DXMatrixIdentity( &m_mtxWorld );

    // Update the frame matrices
    if ( IsLoaded() && UpdateFrames )
    {
        // Reset the bounding box
        m_vecBoundsMin = D3DXVECTOR3(  FLT_MAX,  FLT_MAX,  FLT_MAX );
        m_vecBoundsMax = D3DXVECTOR3( -FLT_MAX, -FLT_MAX, -FLT_MAX );

        UpdateFrameMatrices( m_pFrameRoot,
                             mtxWorld,
                             &m_vecBoundsMin,
                             &m_vecBoundsMax );

    } // End if update frames

}
```

As you can see, the CActor::UpdateFrameMatrices method now supports two additional parameters that allow you to pass in two extent vectors to be filled with the world space bounding box of the passed frame. As the frame we are passing is the root node, and the vectors we are passing are the actor's world space extent vectors, we know that on function return the CActor's m_vecBoundsMin and m_vecBoundsMax vectors will contain the world space bounding box of the root node. This is the world space bounding box that encompasses the entire hierarchy of the actor and all its contained meshes.

## UpdateFrameMatrix - CActor

This is the function where everything comes together. On top of its original job of calculating the world matrices at each frame, it now has to transform the object bounding box of any frame that contains mesh data into world space and return it back to its parent. Because we want the boxes to be calculated after we have returned from traversing the child list, the code for transforming the box will be at the bottom of the function.

The function is now passed pointers two extent vectors which will store the world space bounding box for the current frame being visited in any given recursion of the function.

```
void CActor::UpdateFrameMatrices( LPD3DXFRAME pFrame,
                                  const D3DXMATRIX * pParentMatrix,
                                        D3DXVECTOR3 * pFrameMin /* = NULL */,
                                        D3DXVECTOR3 * pFrameMax /* = NULL */ )
{
    D3DXFRAME_MATRIX * pMtxFrame = (D3DXFRAME_MATRIX*)pFrame;

    D3DXVECTOR3 vecBoundsMin = D3DXVECTOR3(  FLT_MAX,  FLT_MAX,  FLT_MAX );
    D3DXVECTOR3 vecBoundsMax = D3DXVECTOR3( -FLT_MAX, -FLT_MAX, -FLT_MAX );;

    if ( pParentMatrix != NULL)
        D3DXMatrixMultiply( &pMtxFrame->mtxCombined,
                            &pMtxFrame->TransformationMatrix, pParentMatrix);
    else
        pMtxFrame->mtxCombined = pMtxFrame->TransformationMatrix;

    // Move onto sibling frame
    if ( pMtxFrame->pFrameSibling ) UpdateFrameMatrices( pMtxFrame->pFrameSibling,
                                                         pParentMatrix,
                                                         pFrameMin,
                                                         pFrameMax );

    // Move onto first child frame
    if ( pMtxFrame->pFrameFirstChild )
         UpdateFrameMatrices( pMtxFrame->pFrameFirstChild,
                              &pMtxFrame->mtxCombined,
                              &vecBoundsMin,
                              &vecBoundsMax );
```

The first part of the function has hardly changed, with the exception that we now initialize temporary bounding box extents vectors and send them into the child recursions. The sibling recursions all share the same parent, so they are passed the same bounding box vectors that were passed into this function.

When the child list recursions return, the local bounding box extent vectors vecBoundsMin and vecBoundsMax will contain a world space bounding box that encompasses all of the children beneath the current frame. To see why this is the case, we have to look at the new second section of the function that calculates a world space bounding box for any mesh data that might be attached to this frame.

If the current frame's bObjectsBounds member is set to true then it means this frame either has a mesh attached or it influences a skin. In either situation, it means we have a frame that contains a model space bounding box. When this is the case, we transform the model space AABB using the world matrix of the current frame we are visiting.

```
    // Should we also update our actor's bounding box with the new information
    if ( pMtxFrame->bObjectBounds )
    {
        D3DXVECTOR3 vecMin = pMtxFrame->vecObjectMin,
                    vecMax = pMtxFrame->vecObjectMax;

        // Transform the mesh's object bounding box into world space
        MathUtility::TransformAABB( vecMin, vecMax, pMtxFrame->mtxCombined );
```

At this point we now have two world space bounding boxes. The first is the world space bounding box of the mesh data stored at this node (which we have just stored in the frame's world space extent vectors). The second, stored in vecBoundsMin and vecBoundsMax, was returned from the child list and bounds all the children of this frame. We will now merge these two boxes together by adjusting the world space bounding box at this frame so that it also encompasses the bounding box returned from the children.

```
        // Test it against the actor bounding box and update if necessary
        if ( vecMin.x < vecBoundsMin.x ) vecBoundsMin.x = vecMin.x;
        if ( vecMin.y < vecBoundsMin.y ) vecBoundsMin.y = vecMin.y;
        if ( vecMin.z < vecBoundsMin.z ) vecBoundsMin.z = vecMin.z;
        if ( vecMax.x > vecBoundsMax.x ) vecBoundsMax.x = vecMax.x;
        if ( vecMax.y > vecBoundsMax.y ) vecBoundsMax.y = vecMax.y;
        if ( vecMax.z > vecBoundsMax.z ) vecBoundsMax.z = vecMax.z;

    } // End if apply frame's bounding box
```

The frame now contains the world space bounding box for its mesh data stored and all of the meshes stored below it in the tree. Notice in the above conditional code that what we have actually done is grown the temporary bounding box that was returned from the child list to contain the bounding box we calculated for this frame's mesh data. The next line of code describes why we adjusted the local temporary box and not the box stored in the frame.

```
    // Store the bounding box in this frame
    pMtxFrame->vecBoundsMin = vecBoundsMin;
    pMtxFrame->vecBoundsMax = vecBoundsMax;
```

As you can see, the next step is to copy the temporary bounding box which we just calculated into the frame's world space box extents. However, this code is not in a conditional code block and will be executed even if the frame has no mesh data attached. If the frame has no mesh data, then it is simply assigned the world space box that was returned from the child list. This is actually true in both cases, but in the case where we have mesh data assigned, we simply adjust it first by the mesh's world space bounds before assigning it.

This function is also expected to return the bounding box of this frame back to the parent so that the parent can combine the box with its mesh data, and so on back up to the root. As you can see, that last step is to copy this frame's world space bounding box into the passed vector extents so that the parent will have them on function return.

```
    // Test it against the parent's bounding box and update if necessary
    if ( pFrameMin )
    {
        if ( vecBoundsMin.x < pFrameMin->x ) pFrameMin->x = vecBoundsMin.x;
        if ( vecBoundsMin.y < pFrameMin->y ) pFrameMin->y = vecBoundsMin.y;
        if ( vecBoundsMin.z < pFrameMin->z ) pFrameMin->z = vecBoundsMin.z;

    } // End if frame min extents requested

    if ( pFrameMax )
    {
        if ( vecBoundsMax.x > pFrameMax->x ) pFrameMax->x = vecBoundsMax.x;
        if ( vecBoundsMax.y > pFrameMax->y ) pFrameMax->y = vecBoundsMax.y;
        if ( vecBoundsMax.z > pFrameMax->z ) pFrameMax->z = vecBoundsMax.z;

    } // End if frame max extents requested
```

The final bit of this function is not new; it loops through each mesh container stored at the frame and invalidates it. This is important if software skinning is being used as it tells the actor that any skin stored here will have to be re-transformed into world space before being rendered again.

```
    // If this has a mesh container (or set of containers), we must invalidate them
    D3DXMESHCONTAINER_DERIVED * pContainer =
                            (D3DXMESHCONTAINER_DERIVED*)pMtxFrame->pMeshContainer;

    for (;pContainer;
          pContainer = (D3DXMESHCONTAINER_DERIVED*)pContainer->pNextMeshContainer )
    {
        // Flag as invalid
        pContainer->Invalidated = true;

    } // Next Container
}
```

We have now covered all the new actor code that calculates and returns the world space bounding box needed in order to use the actor as a dynamic object in our spatial tree. Of course, by implementing the hierarchy of bounding boxes the way we have, it means that we now have the ability to hierarchically frustum cull an actor even without using our spatial tree system.

For example, imagine you have loaded a single X file containing the entire scene. This scene, with hundreds of meshes and animations would now be contained in a single actor. Every time we update the actor, a world space bounding box is stored at each frame. This box contains the bounding boxes of all frames underneath it (just like a quad-tree or an oct-tree), so you could easily write a function that will traverse the frames of the hierarchy with a camera and perform hierarchical frustum rejection using the techniques we have covered in this chapter (including the optimizations). At each frame, the recursive function could test the world space bounding box of the frame to see if it is within the frustum, if not, the child list does not have to be stepped into possible rejecting hundreds of meshes stored below it. The actor has now become a bounding volume hierarchy (a spatial tree of sorts). You could even run hierarchical bounding volume or ray intersection queries on the actor's meshes if needed. This might be helpful if you wanted to include some form of body part specific damage for example.

# 15.9 Adding Visibility Determination to CTerrain

A few changes have had to be made to the CTerrain class so that it now makes use of the spatial tree. As we know, each CTerrain is made up of a series of CTerrainBlocks where each block is a rectangular mesh that represents a portion of the terrain. When the terrain blocks are created (in the CTerrain::BuildTerrainBlocks method), each terrain block has its world space bounding box registered with the spatial tree as a detail object. This forces the tree to compile a tree that is large enough to encompass the terrain, even though the terrain polygons are never passed into the tree and compiled into the leaves. As discussed in the previous chapter, the last thing we want is for the spatial tree to be only as large as the static data contained within it. It makes no sense for us to compile the terrain triangles into the tree since each block can be rendered more efficiently as a separate mesh. However, we do want the spatial tree to be large enough to encompass the terrain so that a dynamic object that is placed on that terrain is never in a position where it is not in a region of space that does not belong to the tree. This would mean the dynamic object would not exist in any leaf nodes and as such, could not benefit from the visibility system.

Each terrain block will be given a visibility Boolean which will be registered with the spatial tree as a dynamic/tree object. Although the terrain blocks are not literally dynamic (they will never move), by registering a tree object for each terrain block, we can have the terrain block Boolean pointers stored in leaves and set to true by the tree whenever that terrain block is visible. The CTerrain::Render method can then loop through each terrain block and skip rendering any that have not had their Boolean set to true by the spatial tree.

Below we see the CTerrainBlock member variables and have highlighted the two new ones added in Lab Project 14.1.

***Excerpt from CTerrain.h***

```
class CTerrainBlock
{
 public:

    ……Methods Not Shown……
```

```
    // Public Variables for This Class
    ULONG                   m_nVertexCount;     // Number of vertices stored
    UCHAR                  *m_pVertex;          // Simple temporary vertex array
    ULONG                   m_nIndexCount;      // Number of indices stored
    USHORT                 *m_pIndex;           // Simple temporary index array
    LPDIRECT3DVERTEXBUFFER9 m_pVertexBuffer;    // Vertex Buffer to be Rendered
    LPDIRECT3DINDEXBUFFER9  m_pIndexBuffer;     // Index Buffer to be Rendered
    UCHAR                   m_nStride;          // The stride of each vertex
    ULONG                   m_nFVFCode;         // The flexible vertex format code.

    D3DXVECTOR3             m_BoundsMin;        // Bounding box minimum extents
    D3DXVECTOR3             m_BoundsMax;        // Bounding box maximum extents

    long                    m_nTreeObjectIndex;     // SpatialID
    bool                    m_bVisible;             // Is this object visible?

};
```

**long m_nTreeObjectIndex**
This member will contain the spatial ID returned from the ISpatialTree::InsertTreeObject method. This method will be called to register the terrain block with the spatial tree as a dynamic/tree object.

**bool m_bVisible**
This is the Boolean whose address will be stored in the TreeObject when it is registered with the spatial tree. This Boolean will be set to true by the spatial tree during the visibility pass if any leaf in which the terrain block's bounding volume exists is currently visible. The CTerrain::Render function will loop through its array of terrain blocks and only render those which have m_bVisible set to true.

The CTerrain object will also store an ISpatialTree pointer that can be set via a new method called CTerrain::SetSpatialTree. This method simply copies that passed pointer and stores it in a CTerrain member variable. CTerrain::SetSpatialTree is called in the CScene::ProcessEntities method when it encounters a terrain entity and extracts the data to create a new CTerrain object. After the object has been created, the scene's spatial tree pointer will be passed into CTerrain::SetSpatialTree.

The following snippet of code shows a section of the CScene::ProcessEntities function which is called by CScene::LoadSceneFromIWF to extract and process any entities stored in the IWF file. You will recall from our previous discussions that if the entity currently being processed is a terrain entity, its data is extracted into a TERRAINENTITY structure which is then passed to the CTerrain::LoadTerrain method. This method loads the heightmap, textures, and any additional information stored in the passed TERRAINENTITY structure before calling CTerrain::BuildTerrainBlocks to actually create the vertex data for each terrain block.

***Excerpt from CScene::ProcessEntities***

```
If ( THIS IS AN ENTITY WE ARE PROCESSING)
{
   …Snip… : Extract terrain data from file into TERRAINENTTY structure (Terrain)

// Allocate a new terrain object
pNewTerrain = new CTerrain;
if ( !pNewTerrain ) break;
```

```
// Setup the terrain
pNewTerrain->SetD3DDevice( m_pD3DDevice, m_bHardwareTnL );
pNewTerrain->SetTextureFormat( m_TextureFormats );
pNewTerrain->SetRenderMode( GetGameApp()->GetSinglePass() );
pNewTerrain->SetWorldMatrix( (D3DXMATRIX&)pFileEntity->ObjectMatrix );
pNewTerrain->SetDataPath( m_strDataPath );

// Set the spatial tree, to allow the terrain to build detail areas
pNewTerrain->SetSpatialTree( m_pSpatialTree );

// Store it
m_pTerrain[ m_nTerrainCount - 1 ] = pNewTerrain;

// Load the terrain
if ( !pNewTerrain->LoadTerrain( &Terrain ) ) return false;

// Add to the collision system
m_Collision.AddTerrain( pNewTerrain );
}
```

We have snipped out the code that extracts the entity data from the IWF file and stores it in the TERRAINENTITY structure, but once this is done, a new CTerrain block is allocated and its device, render mode, and world matrix are set. Notice the new line (highlighted in bold) which sets the spatial tree pointer of the CTerrain object to the same spatial tree being used by the application. After this, the new terrain object is added to the scene's CTerrain array before its LoadTerrain method is called to actually build the terrain data and all of the terrain blocks. Finally, as we saw several lesson ago, the terrain object is registered with the collision system.

## BuildTerrainBlocks – CTerrain (Updated)

After the terrain information has been loaded from the IWF file and the spatial tree pointer has been set, the CTerrain::LoadTerrain method is called to build the entire terrain and all terrain blocks. This method will call the CTerrain::BuildTerrainBlocks method to build the vertex data for each terrain block.

The next new section of code we will look at is executed in the BuildTerrainBlocks method. It occurs just after the current terrain block being processed has been built. At this point, the spatial tree has not been compiled so we can register the block bounding box as a detail area so that the region of space taken up by this terrain block will be compiled into the tree. The following section of code is actually embedded in a loop that creates each terrain block.

As you can see, we instantiate a TreeDetailArea structure and set its bounding box equal to that of the terrain block. We could store the terrain block pointer in the context pointer but we do not need to. We are only registering this detail area to shape the spatial tree; we do not need to know where it came from.

```
// Building for spatial tree?
if ( m_pSpatialTree )
{
```

```
    TreeDetailArea Area;
    TreeObject     Object;

    // Store detail area bounding box
    Area.BoundsMax = pBlock->m_BoundsMax;
    Area.BoundsMin = pBlock->m_BoundsMin;
    Area.pContext  = NULL;
```

As the bounding box of the terrain block is in model space we should transform it by the CTerrain's world matrix so that it represents the terrain block in world space. We then add the new detail area to the tree to shape the tree during compilation.

```
    // Transform the bounding box into world (tree) space
    MathUtility::TransformAABB( Area.BoundsMin, Area.BoundsMax, m_mtxWorld );

    // Add to the spatial tree
    m_pSpatialTree->AddDetailArea( Area );
```

In order for a terrain block to only be rendered when visible, we will register it as a tree object that has its Boolean pointer pointing at the CTerrain::m_bVisible member. In a moment, you will see that once the tree is built, we will call the ISpatialTree::UpdateTreeObject method to assign the terrain block's Boolean pointer to the leaves in which the terrain block is contained. We cannot do that just yet, since the spatial tree has not been built at this time. All we can do at the moment is create a new tree object in the spatial tree and assign it to point at our terrain block's Boolean pointer. Notice in the following code that our CTerrainBlock class now has an m_nTreeObjectIndex member that is used to store the terrain block's spatial ID.

```
    // Set new tree object details
    Object.pContext  = NULL;
    Object.pbVisible = &pBlock->m_bVisible;

    // Add as a tree object
    pBlock->m_nTreeObjectIndex = m_pSpatialTree->InsertTreeObject( Object );

} // End if spatial tree set
```

That is the only new section of code in the CTerrain::BuildTerrainBlocks method. At this point the terrain blocks is registered with the system but does not yet have its Boolean assigned to any leaves in the tree (no leaves exist yet since the tree has not been compiled).

After CScene::LoadSceneFromIWF has called all the processing methods to extract the IWF data, all objects contained in the file that needed to be created will have been. This means that any terrain entities that were in the file will have already been used to create terrains and the terrain blocks of each terrain will have been registered as detail areas and as dynamic objects.

It is now time to build the spatial tree as we saw in the previous lesson. However, we have an additional step to take with the scene's CTerrain array. Although the terrain blocks are already registered with the spatial tree, we had to wait until the tree was compiled before we could assign the terrain blocks to their relevant leaves. Thus, in the small section of code shown below (at the bottom of

CScene::LoadSceneFromIWF), you can see that after the spatial tree is compiled, we loop through each terrain in the terrain array and call its TreeBuildComplete method (a new method).

***Excerpt from CScene::LoadSceneFromIWF***

```
…SNIP : All Process…… Functions Called Here

// Build the spatial tree and notify the collision engine
if ( !m_pSpatialTree->Build( ) ) return false;

// Notify terrain that objects the spatial tree has been built
for ( i = 0; i < m_nTerrainCount; ++i ) m_pTerrain[i]->TreeBuildComplete();
```

CTerrain::TreeBuildComplete allows the terrain to perform processing after the spatial tree has been compiled. Let us take a look at this simple member function.

## TreeBuildComplete – CTerrain

For each terrain block, this function exacts its model space bounding box and transforms it into world space using the world matrix of the terrain. It then uses the ISpatialTree::UpdateTreeObject method to assign the terrain block to its relevant leaves in the tree by passing in the spatial ID of the terrain block and its world space bounding box. This is no different from how we update the positions of our dynamic CObjects in the CScene::AnimateObjects method. The reason we do it here is so that it only ever has to be performed once. Our terrains will never move or be animated, so we make this a one-off leaf assignment function that is called post-build.

```
void CTerrain::TreeBuildComplete( )
{
    ULONG i;

    // Validate
    if ( !m_pSpatialTree ) return;

    // Render Each block
    for ( i = 0; i < m_nBlockCount; i++ )
    {
        D3DXVECTOR3 vecMin = m_pTerrainBlocks[i]->m_BoundsMin;
        D3DXVECTOR3 vecMax = m_pTerrainBlocks[i]->m_BoundsMax;

        // Transform the bounding box
        MathUtility::TransformAABB( vecMin, vecMax, m_mtxWorld );

        // Ask tree to update so that the block is inserted into it's final leaves
        m_pSpatialTree->UpdateTreeObject( m_pTerrainBlocks[i]->m_nTreeObjectIndex,
                                          vecMin, vecMax );

    } // Next Block
}
```

Finally, although we will not show the code here due to the trivial changes, the CTerrain::Render method has now been updated to only render terrain blocks that have their Booleans set to true. These Booleans will be set to true for any terrain blocks that are found to be in visible leaves when the application issues a call to ISpatialTree::ProcessVisibility.

# Conclusion

In this chapter we have made significant progress with our rendering technologies and our geometry management. We have implemented a rendering system that will perform hierarchical frustum culling and efficiently collect and draw the static geometry in visible leaves. The rendering system in CBaseTree is certainly complex, and to fully understand the code you will probably need several passes through this book and the source code. Fortunately, with this core technology in place, we now have a system that will be carried into the remaining chapters of this course where we will take visibility processing to the next level.

So far, our visibility system has only partly lived up to its name. After all, it does not render only the visible polygons; it actually renders all polygons inside the view frustum, which is not nearly the same thing. If we imagine being in a small room (located within a larger complex) with no windows and maybe a small door, all that would actually be visible to us would be the four walls of the room and maybe a small section of the corridor we can observe from within the room. Using our current system, if we had a frustum with a far plane positioned a long way in the distance, many of the rooms in this example which are not visible would still be rendered since they fall within the frustum and are therefore considered visible in our system implementation. In truth, all we have really done in this chapter is laid the foundation for a proper visibility system. Our final visibility system will be implemented in the next two chapters of this course.

While we have learned how to efficiently render only what is within the camera's FOV, we have not taken into account that some portions of the level will be obscured by others. If our camera is located in front of a huge wall, ideally we should only render the wall. On the other side of that wall there might be a vast terrain or multiple buildings that should never be seen as they are occluded by the wall in front of us. Our current visibility system does not make provisions for this scenario. We have taken only a preliminary step towards determining a proper potential visibility set. That is, we certainly have narrowed things down to a set of polygons that are potentially visible using our frustum tests (and we have definitely rejected quite a lot of polygons that definitely are not visible), but many of those potentially visible polygons can also be removed from consideration with just a bit more work on our part.

The remainder of this course will work towards the goal of building a PVS (Potential Visibility Set) calculator that will pre-calculate at compile time exactly which leaves could possibly be visible from any other leaf in the scene by factoring in the notion of occlusion. PVS systems have been the backbone of commercial computer games since the days of Quake™. They are what allow seemingly massive environments comprised of a very high number of polygons to be rendered at real-time interactive frame rates. With a PVS system in place, the speed at which your game runs is no longer bound purely by the number of polygons in your scene, or even the number of polygons contained in the frustum. Rather, we

are dealing with only the polygons that can be physically seen by the viewer, which is obviously going to be a far smaller number in the general case. If you consider a room with no windows and closed doors, regardless of the size of the level and the number of polygons it contains, our PVS-based system would only have to render the four walls, the floor and the ceiling which the camera can currently see (the hardware clipping can handle any final bits of actual visibility testing within the room). So the next two lessons will open up a wide array of possibilities in terms of the art assets your engine will be able to handle. You will soon be able to draw incredibly high polygon count scenes that would otherwise be unthinkable, even on today's cutting edge hardware.

In the next chapter we will introduce BSP trees as they are going to be the foundation tree for the PVS system we will implement. We will also learn many other techniques that the BSP tree can be used for, such as perfectly sorting alpha polygons and rendering them in the correct order (something that our hash table design from Module I could not guarantee in all cases). We will also discover that BSP trees can be used to carve meshes from one another or fuse multiple meshes into a single mesh. BSP trees are prevalent in computer science and particularly in game development and they will be one of the most useful tools you will learn about in your training as a game developer.

Make sure that you fully understand how our rendering system works before moving on to the next chapter. We will be using this same system again as we move forward with BSP trees (at least initially), so it is important that you devote the necessary time to working through the system. While this was undeniably a fairly complex implementation (especially once the multiple vertex buffer support was introduced), the underlying logic is still quite straightforward. If you take your time and make sure that you are comfortable with the concepts we have covered here, you will be in good shape in the discussions to come.