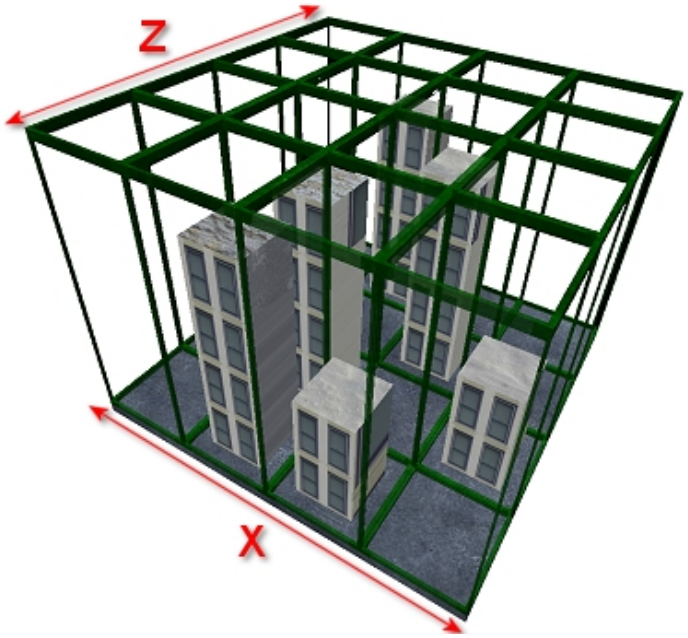
Chapter Fourteen

Hierarchical Spatial Partitioning



Introduction

In this lesson we will study various hierarchical spatial partitioning techniques and use them to implement a broad phase for the collision detection and response system developed in the previous chapter. The techniques we learn in this lesson and the next will also be used to speed up the rejection of non-visible polygon data from the rendering pipeline.

The concepts you learn in this chapter are some of the most important you will ever learn as a game developer, or even as a general programmer. Indeed you will find yourself using them at many points throughout your programming career to optimize a particular task your application has to perform. Although we will apply the tree traversal techniques discussed in this lesson to the area of 3D graphics programming, the usefulness of the ideas we will introduce reach far beyond game creation. For example, such techniques are used by image processing routines to optimize the mapping of true color images to a limited palette of colors. Hierarchical tree structures are also used to perform quick word searches in databases and efficiently sort values into an ascending or descending order. This lesson will be the first of four studying spatial partitioning. It will eventually lead to the implementation of a broad phase for our collision system and an efficient hardware friendly PVS rendering system.

Fortunately, many of the concepts we discuss will not require us to cover a lot of new ground since we already have a fair understanding of the two areas that hierarchical spatial partitioning essentially borrow from: parent/child relationships and bounding volumes. Combined, they allow us to spatially divide our scene into a hierarchically ordered set of bounding volumes that contain all of the scene geometry. These spatial trees can be quickly traversed and entire branches of the tree (along with child bounding volumes and the polygon/mesh data they contain) can be rejected from consideration with only a few simple bounding volume intersection tests.

Over the next few chapters we will be implementing spatial hierarchies in a variety of different flavors. We will implement a base tree class that can be used by the application to interface with all our spatial tree types. We will discuss and implement quad-trees, oct-trees, kD-trees and the famous (or infamous) BSP tree. These four tree types all provide a different way for the scene to be spatially subdivided and each has its uses in certain situations. The BSP tree (Chapters 16 and 17) will be vitally important as it will be used to aid us in the calculation of a Potential Visibility Set (PVS); this is a process that involves pre-computing what polygons can be seen from any point within the level at development time. The calculation of a PVS will allow our application to render scenes of an almost unlimited size (memory permitting) while keeping our frame rates fast and interactive. But before we get ahead of ourselves, let us focus on the basics, since there is a lot to cover.

This chapter will be focused on examining some of the common tree types that are used in games and the theory behind how they partition space and how they can be used. Later in this lesson and in our lab project, we will implement a broad phase for our collision system that will significantly speed up polygon queries on the collision system's geometry database. We will also discuss the usefulness of such systems for rendering, although this will not be implemented or demonstrated until the following lesson where a hardware friendly system will be introduced. Furthermore, in this chapter we will discuss the various utility techniques which typically accompany spatial partitioning algorithms: polygon clipping and T-junction repair, for example. So why don't we get started?

14.1 Introducing Spatial Partitioning

It is difficult to overstate the importance of spatial partitioning in today's computer games. Every game you currently play will undoubtedly use spatial partitioning as the core of its collision system and rendering logic. We learned in Chapter 4 of Module I how the frustum culling of a mesh's bounding volume could be used to prevent its polygon data from being passed through the transformation pipeline unnecessarily. Although the DirectX pipeline does perform a form of frustum rejection on our behalf, it does so at the per-polygon level, and worse still, only after the vertices of that polygon have undergone the calculations needed to transform them into the homogenous clip space.

We discovered that by surrounding a mesh with an axis aligned bounding box, we could test this box against the frustum to introduce a broad phase rejection pass in our mesh rendering logic. If we determined that the mesh's bounding box was outside the frustum, then we knew we did not have to bother passing its polygon data through the transformation pipeline. If that mesh was comprised of 20,000 polygons, we just avoided needlessly transforming them at the cost of a simple AABB/Frustum test. Of course, the same logic can be applied when performing any query on the scene that is required at the per-polygon level, such as we find with collision detection. We know that if our swept sphere does not intersect the bounding volume of a mesh then we do not have to test any of its individual triangles. We might perceive this to be a very simple broad phase implementation when added to our collision system. However, we do not always have our scenes comprised of multiple meshes, such as the internal geometry loaded from an IWF file for example. In such cases the entire scene is represented as a single mesh, so surrounding this with a single AABB would not help at all. That is, it would not allow us to reject portions of that mesh during frustum tests or collision queries. Spatial partitioning is the next level in bounding volume rejection and it works just as well for polygon soups as it does for scenes comprised of individual meshes.

In the first section of this lesson we will bias the demonstrations of spatial partitioning towards performing pure collision queries from the perspective of game physics, such as those discussed in the last chapter. But later we will discuss and implement a rendering system that will also benefit from spatial partitioning while remaining hardware friendly. We will also slant our early discussions toward the spatial partitioning of a polygon soup, although later we will discuss how spatial trees can also be built that partition complete meshes. In fact, the spatial trees we implement will be capable of managing both. That is, we can divide space into bounding volumes that contain both a list of polygons and a list of mesh objects that exist in that area of space. This allows our system to handle cases where the core geometry might be comprised of a huge list of static polygons but the dynamic objects are individual meshes or actors. We want both types to be supported by our tree so that we can quickly reject not only the core scene geometry that lay outside the frustum for example, but any dynamic objects (such as skinned characters) which are also not visible.

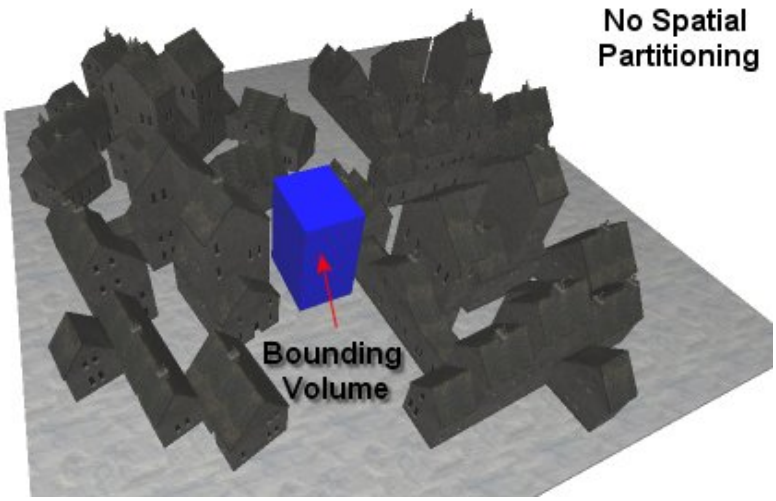


Figure 14.1

Let us begin our journey by imagining a simple town scene (Figure 14.1) consisting of 100,000 polygons. Assume that the blue bounding box in the figure represents an AABB that has been compiled around the source and destination positions that our swept sphere will move between in the current update. We used a similar technique when collecting the terrain polygons for our collision system in the previous chapter. You will recall that rather than trying to test the awkward shape of the swept sphere against the terrain to get a list of potential colliders, we simply built an AABB around the swept sphere and collected any terrain geometry that fell inside it. We will assume that the same logic is being applied here. After all, we are only interested in quickly finding which polygons will possibly intersect our swept sphere so that they can be passed on to the narrow phase. Only polygons that are contained partially or completely inside the bounding volume will need to be tested in the computationally expensive narrow phase of our collision system.

A naïve approach to a broad phase implementation might be to test every polygon in the scene against the AABB surrounding the swept sphere and reject all polygons that do not intersect it. This would certainly work and at the end of the process we would have compiled a list of polygons which intersect the AABB and thus have the potential to be a collider with the swept sphere. This list of potential colliders would then be passed to the narrow phase where the swept sphere intersection tests are performed on each polygon. Remembering that this scene is assumed to be comprised of 100,000 polygons, the pseudo-code to such a broad phase implementation would be as shown below.

> **Note:** We will assume for now that we have a function called 'AABBIntersectPolygon' which determines whether a polygon intersects an axis aligned bounding box. The function in this example is passed the minimum and maximum extents of the axis aligned bounding box and a pointer to the polygon that is to be tested.

```
for (i = 0; i < 100000; i++)                                  // Loop through every poly
{
    CPolygon *Poly = CScenePolygonList[i];                    // Get current poly to test

    if (AABBIntersectPolygon( BoxMin , BoxMax , CPoly ))  // Does it intersect AABB
    {
         PotentialColliderList.Add ( CPoly );                   // Add to List that will
    }                                                           // be passed to narrow
}                                                               // phase

NarrowPhase ( PotentialColliderList );
```

In this example CPolygon is assumed to be the structure used to contain an individual polygon and CScenePolygonList is assumed to be an array that contains the list of polygons comprising the scene. BoxMin and BoxMax are the minimum and maximum extents of an AABB that encompasses the swept sphere's movement in this update. PotentialColliderList is some type of container class that manages a list of polygons and exposes methods allowing us to add polygons to its internal list. Whenever a polygon is found to be intersecting or totally inside the bounding box, it is added to this list. At the end of the loop the broad phase is complete and we have compiled a list of potential colliders. This list is then passed to the narrow phase where each polygon would have to be inspected against the swept sphere as described in the previous chapter.

This is a broad phase implementation to some degree, however any broad phase that requires 100,000 bounding volume tests just to find the potential colliders is obviously not very efficient. When we consider the incredible geometric detail in today's games, testing every individual polygon (even against a simple AABB) every time we wish to perform a query on the scene data is not going to suffice when it comes to performance. The idea of the broad phase is to quickly reject large blocks of polygon data from consideration very quickly. The number of tests needed to find the potential colliders in the broad phase should not come anywhere close the polygon count of the scene as is the case in the above example implementation. We need to be able to say, "My bounding volume is not located within this entire area, so all polygons in this area should be dismissed right away".

So let us start with a naïve approach to spatial partitioning, but one that will serve to solidify certain concepts in less obfuscated way before moving on to the subject of more common spatial hierarchies.

Let us imagine the same scene again, only this time, when the level was first loaded, we built an axis aligned bounding box around the entire level. Let us also imagine that with this information in hand, we divided the area of the scene's bounding box into a 7x7 grid (an example) of bounding boxes as shown in Figure 14.2.

Generating these 49 bounding boxes would be trivial. Once we had the bounding volume of the scene we could just divide its width and depth by 7 giving us a value of N and M for the width and depth deltas, respectively. We would then set up a loop to step through along the width of the scene's bounding box in steps of N and along the depth of the scene's bounding box in steps of M. In the inner loop (M), the coordinate (N*Column, M*Row) describes the minimum extent of the current bounding box being calculated and vector (N*Column + N, M*Row + M) describes its maximum extent.

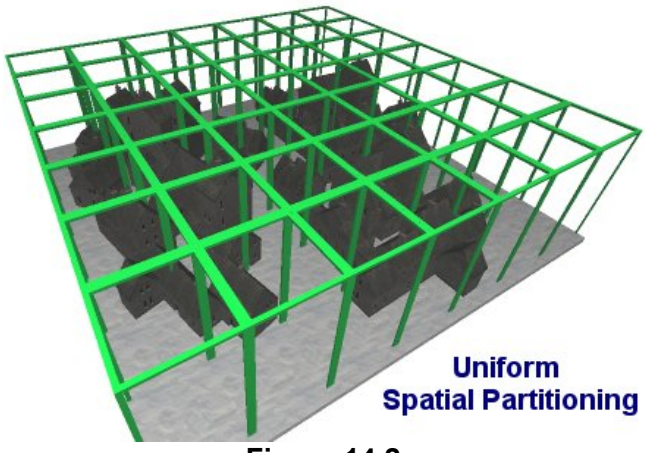


Figure 14.2

We might imagine that in this simple example, we could employ a scene class that contains an array of CBoxNode structures.

Note: The code we discuss in this first section is purely for teaching purposes and will not be used by any of our applications. It is used to solidify concepts. Later in the lesson we will develop the actual code that our applications will use.

```
class CScene
{
    public:
    CScene ( CPolygon **ppPolygons , long PolygonCount);
    CBoxNode  BoxNode [49] ;
};
```

In this very simple example, the scene has a constructor that accepts an array of CPolygon pointers and the number of pointers in this array. This is where we would pass in the array of 100,000 polygons we have loaded from our town file. This function would be responsible for calculating the bounding box of the entire scene and then generating the 7x7 (49) CBoxNode structures.

The CScene class has an array of CBoxNode structures, one for each bounding box we will create to represent the level. This structure might be defined like so:

```
class CBoxNode
{
   public:
   D3DXVECTOR3       m_vecBoxMin;
   D3DXVECTOR3       m_vecBoxMax;
   CPolygonContainer m_PolyContainer;
};
```

This simple class stores the minimum and maximum extents of the bounding box this node will represent, and a polygon container. We might imagine this to be a simple class that wraps an array of polygon pointers and exposes member functions for adding and retrieving polygon pointers to/from that array. We provided many classes that did similar things in Module I and of course, STL vectors could be used for the same task. This container will initially be empty for each box node. We will discuss why we will need this container stored in the box node class in a moment.

The job of the scene's constructor would be to compile a bounding box for the entire scene. This is done by simply testing the vertices of every polygon passed in and recording the maximum and minimum x, y, and z components found. The first section of such a function is shown below.

```
CScene::CScene ( CPolygon **ppPolygons, long PolygonCount )
{
   // Set master scene bounding box to dummy values
   D3DXVECTOR3 vecBoundsMin(  FLT_MAX,  FLT_MAX,  FLT_MAX );
   D3DXVECTOR3 vecBoundsMax( -FLT_MAX, -FLT_MAX, -FLT_MAX );
   int i , k , j;

   // Loop through each polygon in list and calculate bounding box of entire scene
   for ( i = 0;  i < PolygonCount; i++ )
   {
     // Get pointer
     CPolygon * pPoly =  ppPolygons[ i ];
```

```
    // Calculate total scene bounding box.
    for ( k = 0; k < pPoly->m_nVertexCount; k++ )
    {
      // Store info
      CVertex * pVertex = &pPoly->m_pVertex[k];

      // Test if this vertex pierces the current maximum or minimum
      // extents if adjusting extents if necessary
      if ( pVertex->x < vecBoundsMin.x )  vecBoundsMin.x = pVertex->x;
      if ( pVertex->y < vecBoundsMin.y )  vecBoundsMin.y = pVertex->y;
      if ( pVertex->z < vecBoundsMin.z )  vecBoundsMin.z = pVertex->z;
      if ( pVertex->x > vecBoundsMax.x )  vecBoundsMax.x = pVertex->x;
      if ( pVertex->y > vecBoundsMax.y )  vecBoundsMax.y = pVertex->y;
      if ( pVertex->z > vecBoundsMax.z )  vecBoundsMax.z = pVertex->z;

    } // Next Vertex

  } // Next Polygon
```

At this point we have the bounding box for the entire scene. We will now carve this box into 49 boxes (7 rows of 7 columns) and assign the resulting bounding boxes to the 49 CBoxNode objects of the scene class. This code first divides the width and depth of the scene's bounding box (calculated above) by 7 along the X and Z extents to carve the bounding box into 49 sub-boxes in total.

> **Note:** In this example we are subdividing the scene along the X and Z extents, which can be seen if you look at the previous diagram and imagine that we are looking down on the scene from a bird's eye view.

The two loops shown below are used to loop through each row (i) and each column (k) and calculate the index of the current CBoxNode object that we are processing. We use the formula (i * 4) + k. If *i* is the number of rows of boxes that we want (7) and *k* is the number of columns we want on each row (7) then we can see that when *i*=2 and *k*=1 we are calculating the bounding box for the ((2*7)+1) = 15th CBoxNode object in the scene's array. Remembering that arrays use zero-based indexing, we can see how this CBoxNode object would represent the 2nd box (*k*) in the 3rd row (*i*) of our scene.

```
float StepX  = ( vecBoundsMax.x - vecBoundsMin.x)  / 7.0f ; // Seven Columns
float StepZ  = ( vecBoundsMax.z - vecBoundsMax.z)  / 7.0f ; // Seven Rows

for ( i = 0; i < 7; i++ )
{
  for ( k = 0 ; k < 7 ; k++ )
  {
    // Get the index of the Box Node we are currently calculating
    int nBi = ( i * 7 ) + k ;

    BoxNode[ nBi ].m_vecBoxMin.x  =  ( k * StepX );
    BoxNode[ nBi ].m_vecBoxMax.x  =  BoxNode[nBi].m_vecBoxMin.x + StepX;
    BoxNode[ nBi ].m_vecBoxMin.z  =  ( i * StepZ );
    BoxNode[ nBi ].m_vecBoxMax.z  =  BoxNode[nBi].m_vecBoxMin.z + StepZ;
    BoxNode[ nBi ].m_vecBoxMin.y  =  vecBoundsMin.y;
    BoxNode[ nBi ].m_vecBoxMax.y  =  vecBoundsMax.y;
  }
}
```

At this point in the scene's constructor we have uniformly subdivided the bounding volume of the entire scene into 49 separate bounding volumes that together represent the space that was described by the scene's original bounding box. These bounding volumes are now stored in the scene's CBoxNode array.

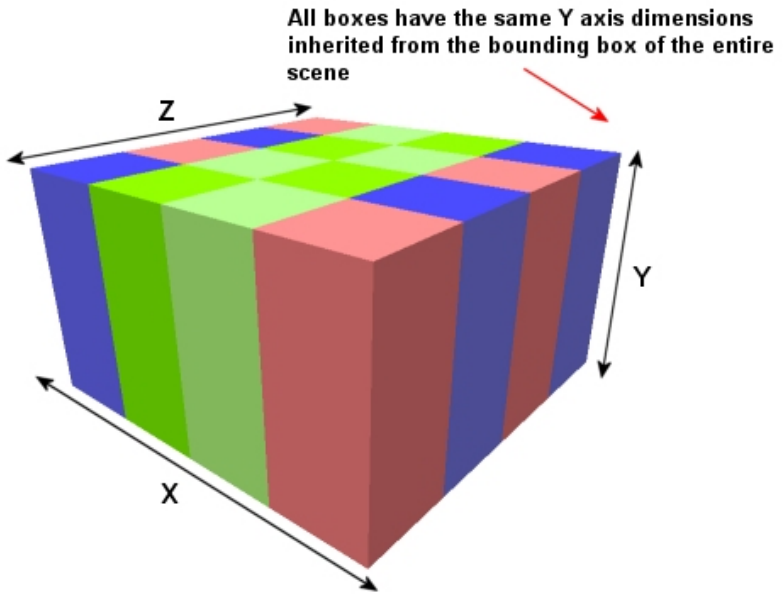


Figure 14.3

Because we are only dividing the scene along the X and Z axes in this example, we assign the minimum and maximum Y axis extents of the bounding box recorded for the entire scene to each of the CBoxNode objects. This means that all 49 sub-boxes generated will all have identical heights along the Y axis. Those Y axis extents are inherited from what we can refer to as the parent bounding box (i.e., the bounding box calculated for the entire scene at the top of the function). Figure 14.3 illustrates the spatial partitioning that has been applied to the scene's original bounding box (albeit for a 4x4 example in this case).

Having 49 empty bounding boxes is fairly useless until the next aspect of our scene preparation is performed. We must now loop through each bounding box and, for each box, loop through each polygon in the scene and test which ones intersect it. For a given bounding box, any polygons that intersect it or are contained completely inside it have their pointers added to the CBoxNode's polygon container. At the end of the following code, our constructor will be complete and when the scene is initialized, we will not only have 49 bounding boxes, in each box node structure we will have a list of polygons that are contained (or partially) contained within that region of space.

```
for ( k = 0 ; k < 49 ; k++ )  // 49 box nodes in this example
{
  // loop through each box
  CBoxNode * pNode = &BoxNode[ k ];

  // Loop through each polygon and test for intersection with box
  for ( j = 0;  j <  PolygonCount; j++ )
  {
    // Get pointer
    CPolygon pPoly =  ppPolygons[ j ];

    // If polygon intersects add its pointer to the box node's container
    if ( AABBIntersectPolygon(   pNode->m_vecBoxMin ,
                                 pNode->m_vecBoxMax
                                 pPoly ))
    {
       pNode->m_PolyContainer.AddPolygonPointer( &pPoly );

   }  // End if Intersect
```

```
    } // End Each Polygon
  } // End Each box
} // End function 'CScene::Constructor'
```

> **Note:** Again you are reminded that this code is not used by any of our applications. It is being used to demonstrate a concept and nothing more. We are using classes with methods here that we have never written, nor do we intend to. Please view these listings as being semi-pseudo code for the time being.

As you can see by examining the final piece of the function code, it involves nothing more than testing each bounding box node to see whether or not any of the scene polygons intersect with it. If so, then a pointer to that polygon is added to the polygon list in the box node. This code assumes that the polygons are stored in each box in a container class that has a member function called AddPolygonPointer, which simply adds the passed pointer to the end of polygon pointer array maintained by the container.

In this particular example, if a polygon is found to be contained in multiple boxes, we store a copy of its pointer in each. This causes no real harm, although when performing subdivision in this way we often wish to make sure that we do not test the same polygon more than once. For example, if we performed a collision query and found that three boxes were intersected by the bounding volume of our swept sphere, we might collect all the polygons from all those boxes and add them to a potential colliders list that is then passed onto the narrow phase. However, it is possible that a large polygon may have intersected all three of those box nodes and as such would be added to the potential colliders list three times. We can certainly work around this in the intersection testing case by testing to see if the given polygon has already been tested for collision so that we do not test it again. However, we must bear in mind that if we have to employ complex loop logic to determine such things, this could possibly outweigh a lot of the savings that the broad phase has introduced.

A common technique when using such a scheme is to allow the polygon structure to store a 'current time' member. When the narrow phase steps through the list, it will process a polygon and set its time member to the current time of the application. If it encounters a polygon in the list with the same time as the current application time, then this must be a pointer to a polygon it has already processed and will skip it. However, this still adds a per-polygon conditional test in the case where the bounding volumes contain polygon data. Under certain circumstances, this can harm performance as well. However, this might be a good strategy for dynamic objects that are assigned to nodes and can have this test performed on the per object level where the conditional tests would introduce negligible overhead. Things are also less simple when spatial partitioning for your rendering pipeline. We desire the vertices of the renderable versions of the geometry to be stored in vertex buffers (static buffers, preferably) rather than CPolygon structures. As such, we have no good way of setting a variable on a per-polygon basis like this and neither would want to. In this case, we may be forced to render the same polygon multiple times.

You will see later that one way to get around this problem is by clipping our scene polygons during the compilation of the spatial tree. In this scenario, a bounding volume will contain only polygons that fit fully inside its volume. If, during the compilation of the bounding volumes, a polygon is found to be inside two boxes at the same time, the polygon will be split into two pieces at the box boundary and the separate fragments will be stored in the respective bounding box. We will also implement an elegant solution for the case when you do not wish clip the polygon data and increase the polygon count of the scene, which will be explained later. For the sake of this current discussion, we will just assume that we

have assigned any polygon spanning multiple nodes to all nodes in which it is contained and will just live with the cost of having it exist multiple times in the potential colliders list.

The kind of spatial subdivision we have shown above would be performed once when the scene is first loaded. Alternatively, the compilation could be done at development time and the scene saved out to file in its subdivided format. Compiling the scene organization at development time is often necessary when we start subdividing scenes with hundreds of thousands or more polygons into thousands of bounding boxes. In these instances, the subdivision of the scene can take quite a long time. Remember, if we are testing hundreds of thousands of polygons against thousands of AABBs, even on today's microprocessors, compilation time will not be instant. If performed at run time when the level is first loaded into the computer's memory, this might subject the user of our application to an unacceptable delay. Therefore, it is often common for a file containing scene data to also contain the information in a subdivided format. Using our simple 49 box subdivided scene, we might store the data in the file such that the file contains 49 bounding box data structures and a list of polygons with accompanying bounding box index values describing which bounding boxes each polygon belongs in. The loading code could simply pull the bounding box data out of the file and use it to create the box nodes. The polygon data could then be extracted and its box index list for each polygon examined and used to assign the polygon to its pre-determined bounding box(es).

14.1.1 Efficient Polygon Queries

Let us now return to the problem of trying to determine which polygons in our level should be considered for the narrow phase during a collision test. Recall that we had assumed the compilation of an AABB around our swept sphere and now wish to calculate the polygons that intersect that bounding box and thus should be added to a potential colliders list. Figure 14.4 reminds us of the current location of our bounding volume within the scene.

We have assumed a level size of 100,000 polygons which each originally had to be tested against the AABB. Now our scene had been divided into 49 bounding boxes and as such, the broad phase requires nothing more than performing an AABB/AABB intersection test between our swept sphere's bounding volume and the 49 bounding volumes of the level. AABB\AABB tests are very cheap to perform and we can narrow down the number of polygons that need to be passed to the narrow phase to just the relative handful of polygons that have been assigned to the bounding boxes that our swept sphere's bounding box intersects.

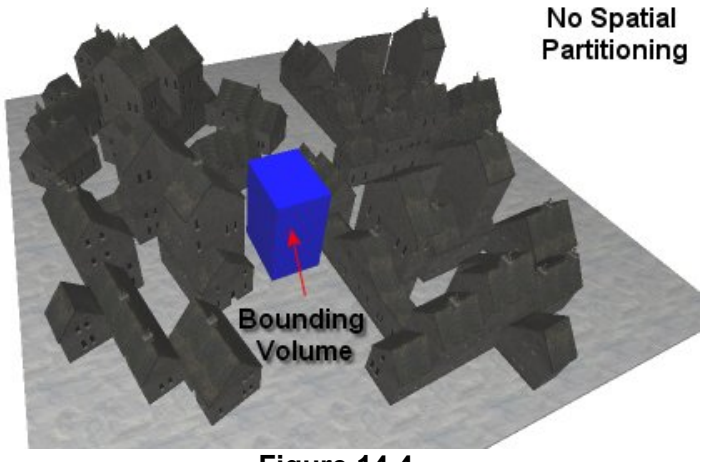


Figure 14.4

As Figure 14.5 demonstrates, after performing only 49 AABB\AABB tests, we find that only two of the scene's bounding volumes intersect our swept sphere's AABB. We can then quickly fetch the polygons assigned to each of these two box nodes and add them to the potential colliders list.

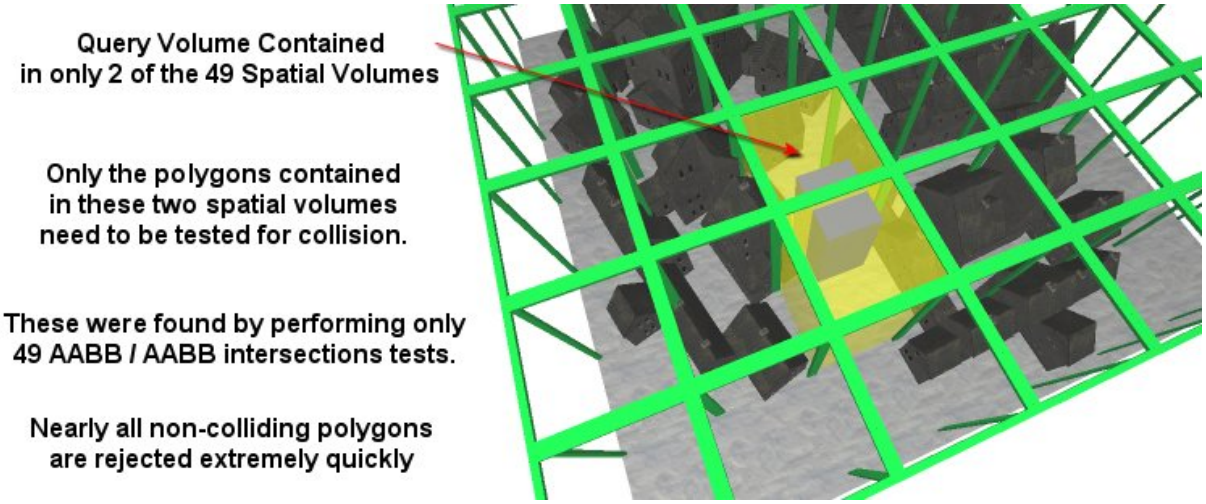


Figure 14.5

If we assumed for the sake of demonstration that the polygons were evenly distributed throughout the level, each box would contain 100,000 / 49 = 2040 polygons.

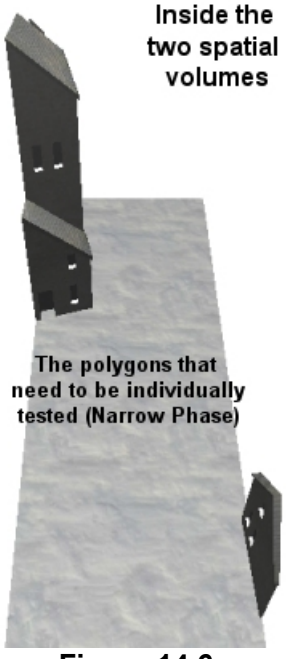


Figure 14.6

Thus with only 49 very efficient AABB/AABB tests, we have just rejected 47 of these boxes and the 2040*46 = 95,880 polygons contained within. Not bad! Only 2*2040 = 4080 polygons would need to be passed to the narrow phase for closer inspection by the collision routines.

As far as the narrow phase is concerned, the world is only as large as those two boxes since these are the only polygons it will need to test. Through the 'eyes' of our narrow phase, the world would look what is shown in Figure 14.6.

Below we see the code to a function that would implement a broad phase in this manner called GetPotentialColliders.

This function could be called by the collision system as its broad phase when the sphere has moved. It would be passed a pointer to the scene and the minimum and maximum extents of the AABB that is bounding the swept sphere. As its final parameter it takes a pointer to an object of type CPolygonContainer. We will assume for now that this is a simple class that encapsulates a polygon array and allows polygon pointers to be added to its array via its member functions. The container is assumed to be empty when this function is called and will be used to store the potential colliders it collects. When the function returns, the collision system will be able to pass this container to the narrow phase.

The function has been deliberately hardcoded for our 49 box example for the purpose of demonstration (again, we will not be using any of this code in practice). It also uses a function called AABBIntersectAABB which is an intersection testing function which we will discuss shortly. The AABBIntersectAABB function takes the extents of the two bounding boxes that are to be tested as its parameters and returns true if the two are intersecting.

```
void GetPotentialColliders( CScene *pScene,
                            D3DXVECTOR3 BoxMin,
                            D3DXVECTOR3 BoxMax,
                            CPolygonContainer *pContainer)
{
     for ( ulong i = 0; i < 49 i++ )
     {
          CBoxNode *pNode = &pScene->BoxNode[i];
          if ( AABBIntersectAABB( BoxMin, BoxMax,
                                  pNode->m_vecBoxMin,
                                  pNode->m_vecBoxMax )
          {
              for ( ulong k = 0; k < pNode->m_PolyContainer.GetSize(); k++ )
              {
                  pPoly = pNode->m_PolyContainer.GetPolygonPointer(k);
                  pContainer->AddPolygonPointer( pPoly );
              }
          }
     }
}
```

As you can see, this function simply tests the bounding volume passed in against the bounding volumes for the scene. If any scene bounding volume is intersecting, the polygons are fetched from the box node's container and added to the potential collider container. When the function returns, the passed container will contain all the polygons that should be more closely examined in the narrow phase.

The application of spatial partitioning is obvious when discussed in the context of running polygon queries such as those often required to be performed by a collision detection system. But does spatially partitioning the scene provide us with any other benefits? Does it possess the ability to speed up rendering as well? It certainly does.

14.1.2 Efficient Frustum Culling

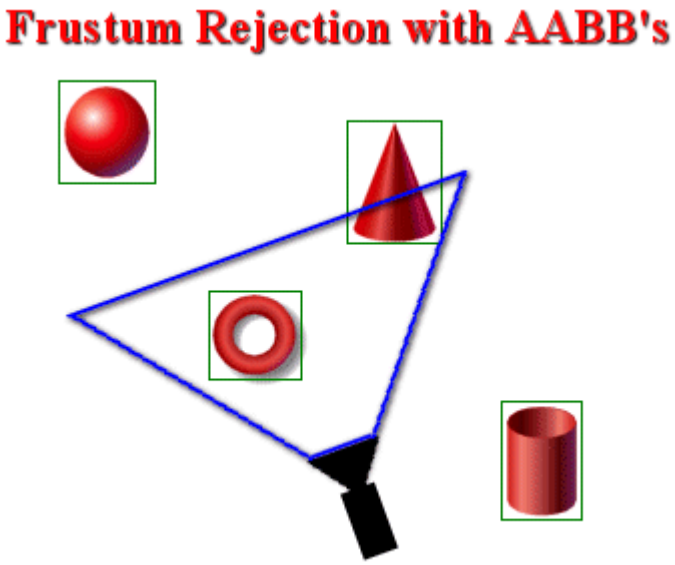


Figure 14.7

In Module I we learned how to perform AABB/frustum culling and used it to reject non-visible meshes prior to being passed through the rendering pipeline. Rather than passing a mesh containing thousands of polygons to the DirectX pipeline and allowing DirectX to determine which polygons are visible and which are not, we decided to help the process along by performing mass polygon culling using bounding boxes.

While the Direct X pipeline is good at what it does and has some very efficient code to reject polygons from the pipeline as soon as they are found to lay outside the view frustum, this culling still has to be performed on a per-polygon basis. If your scene consists of 100,000 polygons, the pipeline will need to perform 100,000 polygon/frustum tests each time the scene is rendered. Even worse is the fact that these polygons all need to be transformed into homogeneous clip space before the culling can commence. While hardware is very fast, this can become a bottleneck with scenes comprised of a large number of polygons.

In Module I we also learned that we could greatly increase our rendering performance if we assign each mesh an AABB and test it against the world space frustum planes prior to rendering that mesh. If we can detect that a mesh's AABB is completely outside the view frustum, we do not have to bother rendering it. Figure 14.7 shows four meshes and a camera along with its frustum. In this example we can see that only two of the four meshes need to be rendered, as the cylinder and the sphere are positioned well outside the camera's frustum.

Let us now return to our little town scene consisting of 100,000 polygons. We are assuming for the time being that this data is loaded as one big polygon soup. It should now be obvious that since we have subdivided our scene into AABBs, the same frustum culling strategy can be employed for our scene's bounding boxes, allowing us to frustum cull geometry in (theoretically) even bigger chunks than the mesh based approach covered in Chapter 4. Any one of our scene's bounding boxes may contain many polygons or perhaps many separate meshes, all of which could be frustum culled from the rendering pipeline with a single AABB / Frustum test.

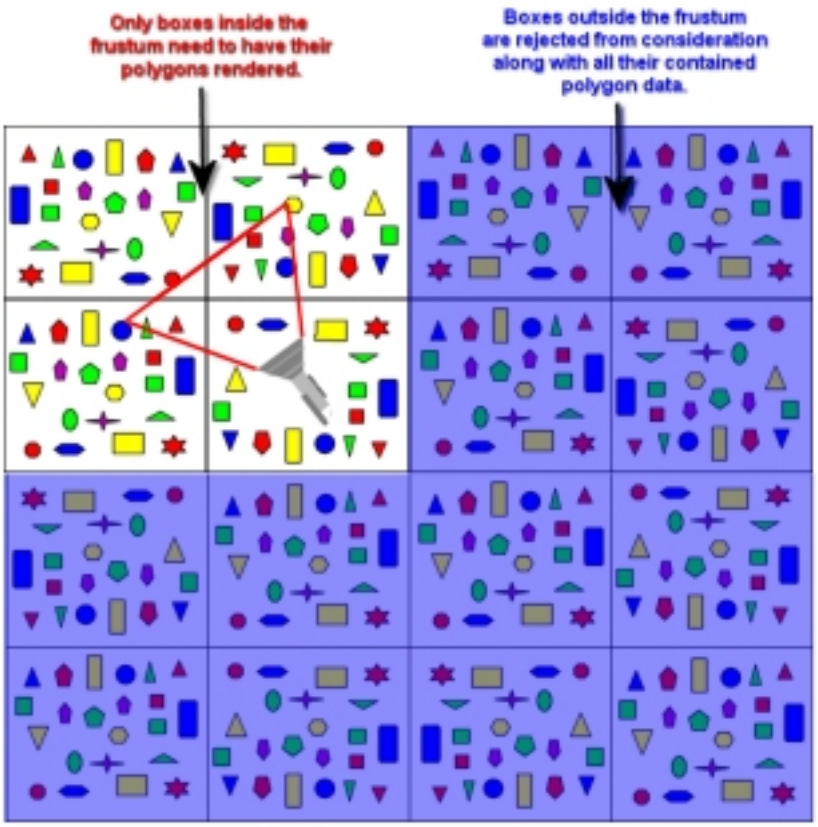


Figure 14.8

Using a 16 box scene example, this time we can see that in Figure 14.8 the camera is positioned and oriented such that only the polygons in 4 of the 16 boxes could possibly be visible. The frustum is shown as the red lines extending out from the camera and we can see that only the four top left boxes intersect the frustum in some way.

Using exactly the same frustum/ AABB code we covered in Chapter 4 of Module I, we have the ability to mass reject quite a bit of scene geometry. If the depicted scene consisted of 100,000 polygons equidistantly spaced such that each of the 16 boxes contained 6250 polygons, we can see that before we render the scene, we would test the 16 AABBs (box nodes) of our scene against the camera's frustum and find 12 of the boxes to be completely outside of it. With just 12 efficient tests, we have rejected 12 * 6250 = 75,000 polygons. Since only four of the boxes intersect the frustum we would only need to transform and render 4 * 6250 = 25,000 polygons ($1/4^{th}$ of the scene).

The following function shows a strategy that might be used to render the 16 box scene shown above. Assuming the scene has already been compiled into its 16 box format and that we are using an AABB / Frustum function which is a member of the camera class called IsAABBInFrustum (the code of which was covered in Module I), the rendering strategy might look something like this:

```
void CScene::Render( CCamera *pCamera )
{
      // Loop through each box
      for ( int i = 0 ; i < 16 ; i++ )
      {
         // Is this box inside the frustum or partially inside the frustum
         if ( pCamera->IsAABBInFrustum( &BoxNode[i].m_vecBoxMin ,
                                        &BoxNode[I].m_vecBoxMin ) )
         {
            // It's a visible box so render all the polygons stored here
            for ( k = 0 ; k < BoxNode[i].m_PolyContainer.GetPolygonCount(); k++ )
            {
                  // Get a point to the current polygon we wish to test
                  CPolygon * pPoly = BoxNode[i].m_PolyContainer.GetPolygon( k );

                  // Render Polygon
```

```
                    RenderPolygon ( pPoly );
                }
            }
        }
}
```

Of course, the above code is for example only and you certainly would not want to render all the polygons assigned to a given box one at a time as that would negate any performance gained from batch rendering. Later we will implement a system that will provide our code the benefits of spatial rejection during rendering but do so in a way that makes it hardware efficient (in fact, this will be the main focus of the next lesson). Hopefully though, even this simple code demonstrates the benefits of spatial partitioning, not only for performing per-polygon queries on the scene's data set such as intersection testing, but also for efficiently rendering that dataset when much of the scene lay outside the frustum.

Indeed it may have occurred to you as you were considering this concept that frustum culling is simply another form of intersection testing. That is, we are colliding one volume (our frustum) with some other volume (the boxes) to collect a set of potentially *visible* polygons. Those polygons are then passed on to a narrow phase (the DirectX pipeline) where the actual visible polygons will be rendered and the others fully or partially clipped away.

Finally, it should be noted that this same system can be used whether your scene is represented as a single mesh or comprised of multiple meshes or both. For example, a box node could contain a list of mesh pointers instead of a list of polygons. Assigning a mesh to bounding boxes could be done by simply figuring out which scene bounding boxes it intersects during the compile process. Later on, we will provide support for both of these systems. That is, the static geometry we load from an IWF file will be spatially partitioned and stored in bounding volumes at the per-polygon level, while dynamic objects such as actors will be assigned to bounding volumes at the object level. In our case, the nodes of our spatial tree will have an understanding of what polygons are contained inside them and our dynamic/detail objects will have knowledge of which spatial nodes they are contained within (more on this later).

Although we have seen how vital spatial partitioning is and how it can improve polygons querying efficiency, the system described so far is really quite inadequate. We have been using very simple numbers to demonstrate the point, but in reality, game scenes tend to be quite large and we ultimately wish to send as few polygons to the narrow/rendering phases as possible. This means that we generally wish to have far fewer polygons stored in each box node. Logically this also means subdividing the scene into many more box nodes. It is more probable that we would want to divide the average scene into thousands of boxes so that each box only contains a handful of polygons. This would mean for example, that if our swept sphere's AABB was found to be intersecting two box nodes, we would only be sending a handful of polygons to the narrow phase, instead of 4000 as in our previous example.

> **Note:** To be fair, given the performance of today's rendering hardware, larger node sizes are not necessarily going to be a bad thing. However, for CPU oriented tasks like collision detection, smaller batch sizes are definitely preferable.

Now that we basically understand what spatial portioning is and how bounding volumes can have scene polygon/mesh data assigned to them, we can take this conversation to the next level and discuss hierarchical spatial partitioning.

14.2 Hierarchical Spatial Partitioning

While the strategy discussed in the last section was great for demonstrating the basic process, it left much to be desired. In our previous example we simply used the value 49 as the number of boxes to divide our scene into, but in reality you would probably want to divide your scene into many more regions than this. If a level contained 100,000 polygons for example, each box would contain 2040 polygons. Even if the query volume only intersected a single box, that is still far too many polygons to send to the narrow phase of our collision system. Remember, given all that must be accomplished in any given game frame, our polygon queries will need to be executed within mere microseconds if frame rates are to remain acceptable.

It would seem that in order to make this strategy more efficient, we would want to divide a typical scene into many more boxes than this. For example, we might decide that we do not want any more than 100 polygons to be contained within a single box node. This would require the scene to be divided into 1000 boxes in our previous example. While this obviously means the initial scene would take much longer to compile (as we would now have to test each polygon against 1000 AABBs instead of just 49) this is not a problem since this can be done as a pre-process at development time and the information saved out to file as discussed earlier. The real problem now is that the broad phase has now been made much more expensive because it would have to test 1000 AABBs instead of just 49 to collect potential colliders for the narrow phase. This is obviously going to be quite a good bit slower. Also, it means that the rejection of a single box now only rejects 100 polygons instead of 2040 polygons, so each AABB/AABB test has less bulk rejection impact. On the bright side, after we have performed all the AABB/AABB tests in the broad phase, we would be left with a considerably smaller polygon list that needs to be passed to the narrow phase.

So while the dividing of the scene into many more smaller boxes has reduced the number of potential colliders that are collected and passed to the narrow phase of our collision system (or rendered in the case of frustum culling), much of CPU savings is lost to the larger number of AABB/AABB tests we have to perform. To be sure, this would still be much faster than just passing every polygon to the narrow phase, but unfortunately it is still nowhere near fast enough.

So how can we divide the scene into many more boxes with small clusters of polygons stored in each box while still maintaining the ability to reject the individual boxes from queries extremely quickly? We use a spatial hierarchy.

We discussed hierarchies in Chapter 9 when we learned how to load in hierarchical X files. The hierarchies in that chapter established spatial relationships between meshes in an X file. The hierarchy could be thought of as a tree structure where each node of the tree was a D3DXFRAME which had pointers (or branches) to other sibling and child frames in the tree. Some of the frames actually contained meshes (via mesh containers) and using the tree analogy, we can think of these nodes as being the leaves of the tree. When we examined the D3DXFrame hierarchy, the parent-child relationship between the frames in the tree established how meshes in the scene were related. If rotation was applied to a node in the tree, the children of that node also inherited the rotational change. This allowed us to store a car model as separate meshes, where the wheel meshes could be rotated independently from the car body stored in a parent node (frame). Any movement or rotation applied to the car body node was

also inherited by the wheel nodes so that when the car body was moved, the wheels always moved with the car body and maintained their correct spatial relationship.

Hierarchies are used so often in game development and in so many different areas that it is hard to imagine creating a 3D application without them. Spatial partitioning data can also be represented as a hierarchy which makes finding the regions of the scene that need to be queried or rendered extremely efficient. If we think back to our D3DXFrame hierarchy from Chapter 9, we can easily imagine that we could assign each frame in the hierarchy a bounding box which was large enough to contain all the meshes below it in the hierarchy (i.e., meshes belonging to direct and indirect child frames). If we were to do this, then the frame hierarchy could be traversed from the root node down during a frustum culling pass and as soon as a frame was found that had a bounding box that was situated outside of the frustum, we could stop traversing down that section of the hierarchy and reject the frame along with all of its descendants from rendering consideration. This rejected frame may have had many child frames which each had many child frames of their own with meshes attached. By simply rejecting the parent frame of the hierarchy, we reject all of the children and grandchildren, etc. with one simple test at the parent node. Of course, this strategy assumes that the parent node's bounding volume has been calculated such that it encompasses the bounding volumes of all its child nodes, and so on right down the hierarchy (i.e., bounding volumes are propagated *up* through the tree, getting larger as more children join in).

It might have occurred to you that this same hierarchical technique could be employed with spatial partitioning. Rather than simply divide the scene into 49 bounding boxes and store all 49 boxes in the scene class, we could instead recursively divide the level, creating a tree or hierarchy of bounding volumes. In this example, the bounding volume being used is an AABB, but we could use other bounding volumes also (spheres are another popular choice).

The following images demonstrate the division of our same example scene hierarchically. The end result in this example is actually a scene that will be divided into 64 bounding boxes (at the leaves of the tree). Because these bounding volumes are connected in a tree like structure, queries can be done extremely quickly.

Forgetting about how we might represent this subdivision technique in code for the moment, let us just analyze the properties of the images. The scene is as before, a large rectangular region of polygon data. The bounding box of this entire region would be considered the bounding box of the entire scene.

When the scene constructor is called we might imagine calculating this bounding box node and storing it in the scene class. This will be the root node of the tree and the only node pointer the scene will store. This concept is not new to us. Recall that our CActor class stores only the root frame of the hierarchy. The rest of the frames in the hierarchy can be reached by traversing the tree from the root. We are now imagining a similar relationship between our spatial nodes. To accommodate this arrangement we can upgrade BoxNode to store pointers to child BoxNode structures.

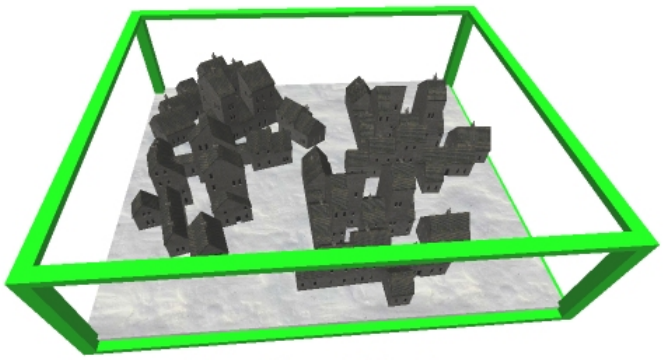


Figure 14.9

```
class CBoxNode
{
   public:
   D3DXVECTOR3       m_vecBoxMin;
   D3DXVECTOR3       m_vecBoxMax;
   CPolygonContainer m_PolyContainer;
   CBoxNode *        m_Children[4];
};
```

Notice that we have now added an array of four CBoxNode pointers to the CBoxNode class. This means each node will have four child nodes.

> **Note:** A spatial tree that divides the parent node space into four at each node is called a *quad-tree*. Thus, what we are looking at above is a Quad-tree node. You will discover later how the only real difference between an oct-tree, quad-tree, and kD-tree is simply how many children each node in the tree has. For the sake of explaining hierarchical spatial partitioning in this section, we will use a Quad-tree in our examples.

As Figure 14.9 shows, the first step in compiling our spatial tree is to compute the bounding box of the entire scene. The bounds calculated will be stored in a CBoxNode structure whose pointer is stored in the scene class. This will be the root node of our tree.

In our example we will be using a quad-tree to partition space which essentially means each node in our tree has four direct child nodes and therefore, at any given node in the tree, its immediate children partition its bounding volume into four equal sub-volumes (called *quadrants*).

With the root node created, we enter a recursive process to build the rest of the tree. At each recursive step (starting at the root) we are passed a list of polygons contained in that node's volume and have to decide whether or not we wish to partition this space any further. If we do decide that the bounding volume of the current node is sufficiently small or that the number of polygons that have made it into this volume are so low in number that further spatial subdivision of this region would be unnecessary, we can simply add the polygons in the list to the node's polygon container and return. We do not bother creating any children for this node. It is essentially a node at the end of a branch and is therefore referred to as a *leaf node*. Leaf nodes are the nodes at the ends of branches which contain polygon/mesh data. The branch nodes are sometimes called *normal nodes* or *internal nodes* or *inner nodes* or simply just nodes (versus leaves).

However, if we do wish to further partition the space represented by the node's bounding box, we divide the node's bounding box into four sub-boxes. We then create four child nodes and assign them the bounding boxes we have just calculated. We then classify all the polygons in the list of polygons that made it to this node against the four bounding boxes of the child nodes. This allows us to create four sub-polygon lists (one per child). We then recur into each of the four child nodes, sending it its polygon list so that this same process can occur all the way down the tree. Notice that when a node has children, it stores no polygon data (at least in this vanilla implementation) and is considered to be a normal node (as opposed to a leaf node at the end of a branch of the tree).

In Figure 14.9 we generated the root node and its bounding volume. Let us step through the process of building the quad-tree with images to solidify our understanding. Please note that in these examples we are still only carving up the scene into 49 boxes at lowest level of the tree since this makes the images much easier to decipher. You can imagine however that this same process can be performed to divide the scene into 1000s of boxes which are hierarchically arranged.

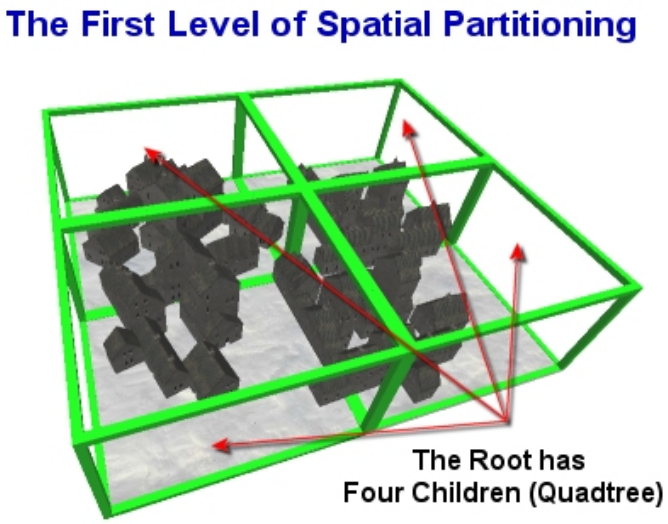


Figure 14.10

We would first process the root node, being passed the list of polygons for the entire scene. We would determine that the node's bounding box is very large and therefore its space should be further partitioned. The polygon list would contain a large number of polygons and we will not be creating a leaf at this node. Remember, we generally want to store only a handful of polygons in the leaf nodes so that we minimize the number of polygons that are collected and sent to the narrow phase when a leaf node's volume is intersected.

Thus, we create four child nodes and attach them to the root node. We would then divide the bounding box of the root node uniformly into four sub-boxes, each of which describes a quadrant of the parent volume (Figure 14.10). Each child node would be assigned one of these child bounding volumes. The list of polygons would then be tested against the four bounding volumes and a list compiled of the polygons contained in each box (four lists). It is at this stage that any polygons found to be spanning a box boundary could be clipped so that the polygon list compiled for each child will contain only the exact polygons that are contained inside its volume. If clipping is being used, a child node will never be sent a polygon list that contains polygons that span the boundary of its bounding box. We would then recur into each child passing in the polygon list that was compiled for it by the parent.

Remembering that this is a recursive process, Figure 14.11 shows what our tree would look like after we have stepped into each child of the root and subdivide their space into four, with polygon lists created for each child node. In reality, the repeated subdivision of each child branch will typically be performed at one time, but we took a bit of artistic license here to demonstrate the subdivision.

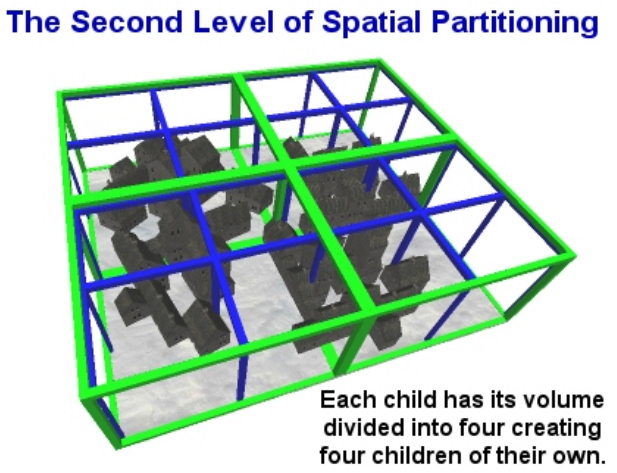


Figure 14.11

If the children of the root node's children are where our recursive process was to end, Figure 14.11 would show the total level of subdivision. As we can see, the root node's bounding box have been divided into four quadrants representing the volume of its four immediate children. Then, each child node of the root, has four children of its own which further partition its volume into four more sub volumes. The blue bounding boxes in Figure 14.11 show the bounding volumes for nodes positioned at the 3rd level in the hierarchy. If we decided to stop partitioning here, then each blue box would be a leaf node and would store a list of polygons that are contained within its volume; the nodes at the first and second level would not. They would just store a bounding volume and pointer to the child nodes. However, in this example we will not stop partitioning at the 3rd level, but will stop at the 4th level instead.

This means that each blue box node in Figure 14.11 would also have four children of its own as shown by the red boxes in Figure 14.12. In our example we are going to stop subdividing here at the 4th level in the hierarchy (3rd level of partitioning). This means, the 4th level of our hierarchy will contain the leaf nodes where the polygon data will be stored.

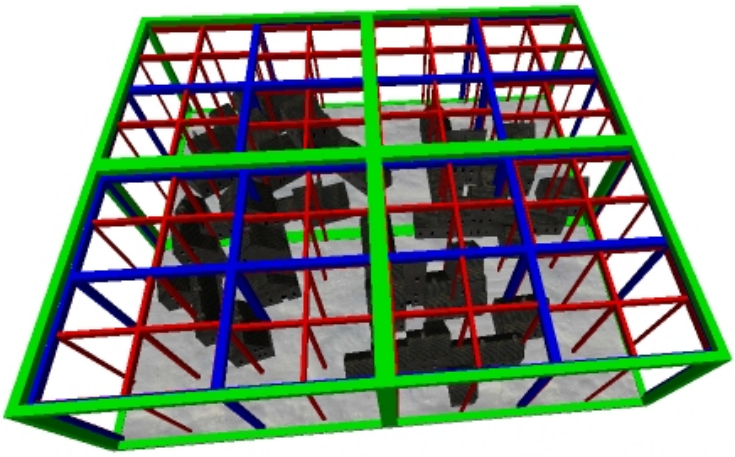


Figure 14.12

Of course, in reality we would subdivide this level to a much greater degree than just 64 bounding boxes, but this subdivision is easy to illustrate in diagrams and subsequently discuss. An actual level that contained 100,000+ polygons would need to be partitioned into hundreds of bounding boxes in order to get optimal collision query performance.

It is also worth noting that in our example we are uniformly partitioning space to get certain sized bounding volumes at the leaf nodes. When this is the case, the quad-tree built will be balanced and every child node at a certain level of the tree will have the same number of children. This is because the scene would be uniformly divided into equal sized leaf nodes. However, often we will not want to subdivide space further if, at any level in the hierarchy, a node contains no polygon data. When this is the case we will generally just make the node an empty leaf (i.e., a node with no children or polygon data). An example of this is shown in Figure 14.13

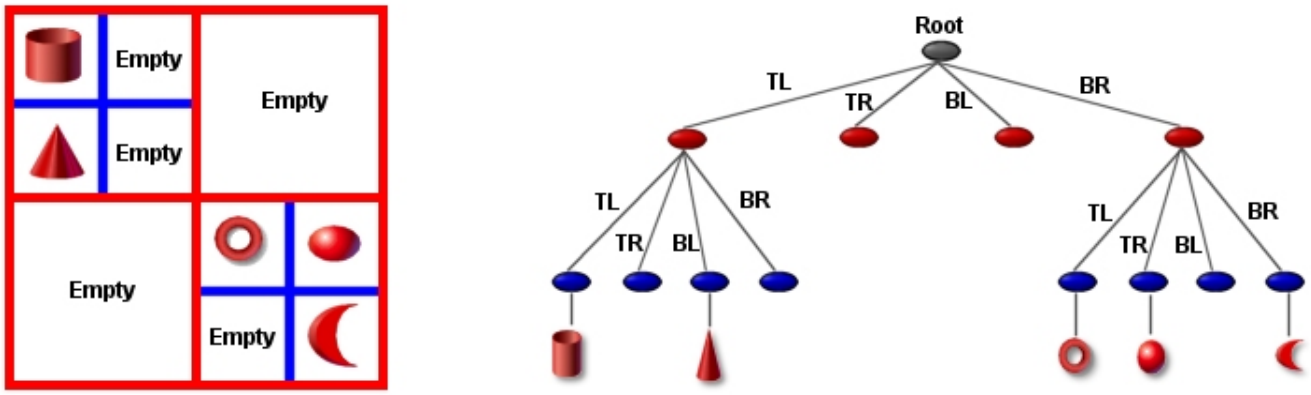


Figure 14.13

On the left side of the image we see the top down view of some shapes being compiled into a quad-tree. At the root node we store the entire bounding box which is then divided into four child nodes representing the four quadrants of the original bounding box. We can see that when building the second level of the hierarchy, the top left and bottom right child of the root contain polygon data and therefore, they are further subdivided (we step into the child nodes and continue our recursion). However, the top right and bottom left children of the root node have no polygon data contained in them so we will not do anything here and return. The fact that we have not created any children for these nodes makes them leaf nodes. Since they have no polygon data, they are empty leaf nodes. On the right of the image we show the shape of the quad-tree. The root has four children and two of those have four children of their own. At the third level, subdivision stops and some of the nodes at this level are empty leaf nodes and some are actual leaf nodes containing polygon data. Notice that we also have empty leaf nodes in the second level too (TR and BL). It would be wasteful in most circumstances to subdivide empty space as these empty volumes will have to be traversed and tested when collision queries are made even though we know that no polygon data will ever be found. The spatial tree shown in Figure 14.13 is not perfectly balanced as the leaf nodes are not all contained at the same level in the hierarchy. Does this matter?

Ideally it would be nice if our tree could be completely balanced so the situation does not arise where some queries may take substantially longer to perform than others merely because the queries are being performed in a section of the tree where the leaf nodes go very deep and more nodes must be traversed before the queried data is located. In a perfectly balanced tree where all leaf nodes existed in the same level of the tree, scene queries would take almost identical times to execute, giving us a consistent query time. We would also use a consistent amount of stack space during the traversal. However, while a balanced tree is a nice thing to achieve, that does not mean we should needlessly subdivide sparse regions of the scene simply to push the sparsely populated or empty leaf nodes down to a uniform level (i.e., alongside leaves from densely populated regions of the scene). After all, making the tree deeper than it need be in many places would essentially be forcing all tree queries to operate under the worst case performance scenario. Tree balance is a more important factor to consider when dealing with kD-trees or BSP trees, as we will discuss in time, but is still something you should bear in mind with all tree implementations. If you find that your scene is taking much longer to query in some area versus others, it may be a balance issue, and in some cases the tree can be made shallower in the troubled areas by not subdividing to such a large degree. It is generally preferable to have you frame rate run consistently at

50 frames per second rather than being at 100 frames per second in some regions and 10 frames per second in others. So consistent query times are certainly desirable where possible.

Using the example quad-tree generated in Figure 14.12, let us see how we might efficiently query the tree for collision information. Once again, at the moment our focus is very much on the implementation of a broad phase for our collision system. We will discuss using spatial trees to speed up rendering later in this lesson and flesh this theory out in the next lesson with a hardware friendly rendering solution.

14.2.1 Hierarchical Queries using Spatial Trees

For this demonstration we will once again assume that we are testing an AABB (surrounding the swept sphere) against the tree starting at the root node. We are interested in finding any polygons in the level that have the potential to collide with the swept sphere. Ideally, we wish to reject all non-potential colliders from consideration as quickly as possible.

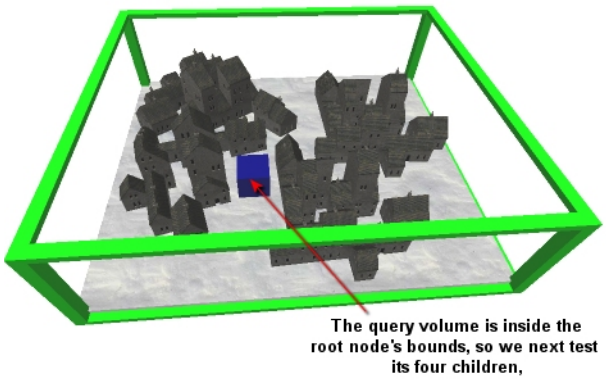
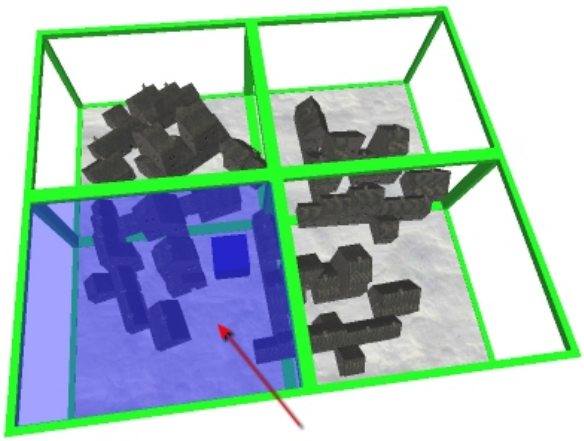


Figure 14.14

The query function would be a simple recursive function which steps through the tree performing intersection tests between the query volume and the bounding boxes of the child nodes of the current node being tested. When an intersection occurs with a child node, the function recursively calls itself to step into that child and continues down the tree. So whenever we find a child node that does not intersect the query volume, we can immediately reject it and all its child nodes (and all the data they contain) instantly.

If at any point we visit a node that has polygon data, we add the data contained in that leaf node to the container being used to collect the polygons for the narrow phase. In Figure 14.14 we show our query volume positioned somewhere within the root node's bounding volume. The first thing we would do is test this bounding volume against the bounding volumes of the root node's four children.



Query volume is determined to be inside the bounds of only one of the four children. Three children and 3/4 of the scene instantly rejected from further consideration.

Figure 14.15

As Figure 14.15 illustrates, the query volume would be found to be inside only one of the bounding volumes of the root's children. In this example we can see that it is contained in only the bottom left child node. At this point, we can ignore the other child nodes completely and they do not have to be further traversed. So with four AABB/AABB tests we have just rejected ¾ of the entire scene (assuming even polygon distribution). We know that nothing in the other three child nodes can possibly intersect our query volume, so we ignore those child nodes and their children, and their children, and so on, right down that branch of the tree. If we assume that our level contains 100,000 evenly distributed polygons, we have just rejected 75,000 of them with four simple AABB/AABB tests. Bear in mind that we are only performing very light spatial subdivision here (four levels), but in reality, that step would have probably rejected 100's if not 1000's of leaf nodes and all the polygon data contained within them.

In this instance of the recursive query routine we discovered that the bottom left child of the root was intersected, so the function will now traverse into that node. In the next step we do the same thing all over again. That is, we test the query volume against the four child nodes of the bottom left child of the root. The children of the bottom left child of the root are shown as the blue boxes in Figure 14.16. As you can see, we discover an intersection with only one of the children and the other three child nodes can be ignored. In the first step we whittled away ¾ of the scene, in this second step, we have whittled it down to just a ¼ of that again. With just eight AABB tests, we have rejected 15/16^{ths} of the entire scene. Not bad at all.

Query volume is determined to be inside the bounds of only one of the four children. Three children and 3/4 of the remaining scene instantly rejected from further consideration.

Figure 14.16

At this point we are located at the bottom left child of the root and have determined that only its top right child is intersected by our query volume. Therefore, the function would call itself recursively again stepping into this child (the highlighted blue box in Figure 14.16)

Next we would step into the top right blue child (Figure 14.16) and would see that it too has four children of its own. Once again, we would test the query volume against its four children (the four red boxes highlighted blue in Figure 14.17) and determine that the query volume is contained in only two of them. When we traverse down into both of these child nodes we find that they have no children and we have reached the leaf nodes of the tree. We then collect the polygon/mesh data stored in those two leaves and add them to the potential colliders container. We can now return from the function, unwinding the call stack back up to the initial function call made to the root by the application. We would have a container filled with only the potential colliders which can then be passed into the narrow phase. We achieved our goal with only 12 AABB tests. Imagine the savings when the geometry has been divided many levels deep and has created 100's of leaf nodes instead of just 64. For example, if the scene was compiled had 1000 leaf nodes, the first four AABB tests at the root would have instantly rejected 750 of those leaves. This is exactly why hierarchical spatial subdivision is so powerful.

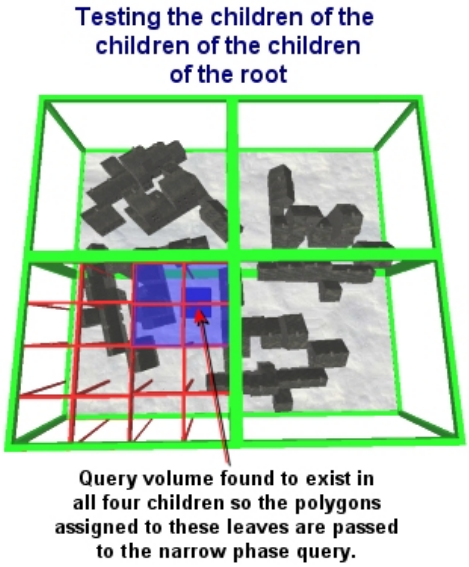


Figure 14.17

14.2.2 Hierarchical Frustum Culling

Let us quickly discuss scene rendering as another example of the benefits of hierarchies. This will only be lightly discussed in this lesson since implementing a hardware friendly rendering system for our spatial trees will be the core subject of the next lesson.

Figure 14.18 shows the first phase of the rendering of this spatial hierarchy. Starting at the root node and traversing down the tree, we can see that at the root node we test its four child nodes (the four quadrants of the entire scene) against the view frustum using AABB / Frustum tests. After these four simple tests we can see that only one of the child nodes of the root intersects the frustum, so the other three child nodes are ignored along with all their children. We have just rejected ¾ of our scene's polygon data from being rendered with these four tests.

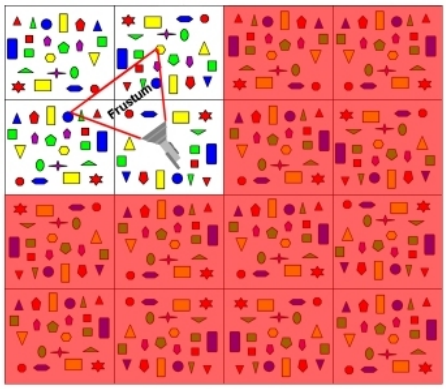


Figure 14.18

The top left child of the root does contain the frustum however, so we will need step down into this node and narrow the set further by testing its four child nodes against the view frustum.

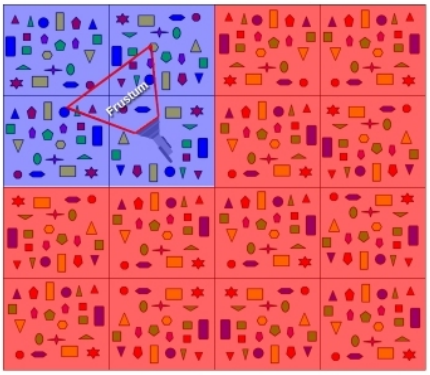


Figure 14.19

When we visit the top left child of the root we must then test to see whether any of that node's child nodes are within the frustum. Once again, we test the frustum against the four bounding boxes of the child nodes, which are shown in Figure 14.19 as the four blue boxes.

One thing that should be obvious looking at Figure 14.19 is that at this level in the tree, all four of the blue child nodes are partially inside the frustum, so we are unable to reject any child nodes at this point. All four child nodes of the top left child of the root will need to be visited to further refine the polygon set that needs to be rendered.

For each of the four blue nodes, we step down and visit their four child nodes. One at a time we determine which of its child nodes have bounding boxes that intersect the frustum. In our simple example scene (Figure 14.20), the children of the blue nodes are also the leaf nodes at the bottom of the tree. These leaf nodes contain the polygon data, so once we find that a leaf node is inside the frustum, we can render the polygons contained there. Alternatively, if you are doing a deferred rendering pass (such as storing the polygons in a queue for sorting purposes) then these are the polygons that should be collected and added to your rendering list.

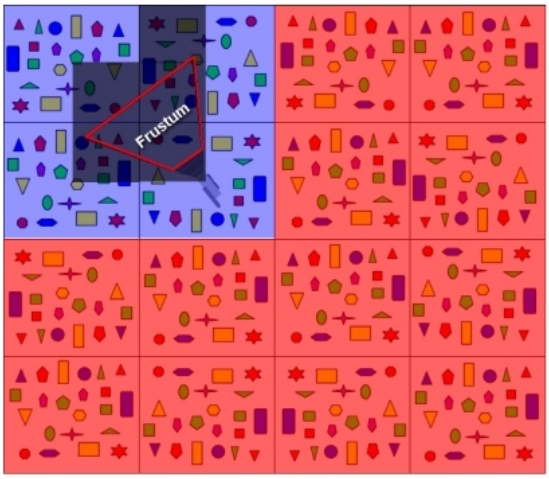


Figure 14.20

As we visit each blue node we determine which of its children are inside the frustum and need to be rendered. We can see in Figure 14.20 that if we start by visiting the top left blue node, only its bottom right child intersects the frustum and it is the only one of its four leaf nodes that needs to be rendered. Then we visit the top right blue node and determine that its top left and bottom left child nodes partially intersect the frustum and need to be rendered. When we visit the bottom left blue node, we see that only one of its four child leaf nodes intersect the frustum (its top right child node). Finally, when we visit the final blue child node on the bottom right, we see that the only child leaf node that is visible is its top left leaf node.

Thus, with the camera positioned as shown in this diagram, only five leaf nodes from a scene comprised of sixty four leaf nodes need to be rendered. By stepping through the hierarchy and performing 24 AABB / Frustum intersection tests, we have rejected 92% of our scene polygons from having to be rendered.

Unfortunately, coming up with a hardware friendly rendering solution is not as trivial as it may appear to be. Even with such large scale rejection of polygon data at our disposal, the strategy of collecting the visible polygons and rendering them must be well thought out so that CPU burden is kept to a minimum. On modern graphics cards, the GPU is very, very fast and you may easily find cases where brute force rendering beats a naïve implementation of hierarchical spatial partitioning. A system that burdens the

CPU will suffer greatly when compared to its brute force counterpart when the entire scene is contained inside the frustum and nothing can be culled. If the entire scene is contained inside the frustum, then everything will need to be rendered anyway. In that case, brute force will always be faster due to the fact that it does not need to perform any tree traversals. So what are our options?

One implementation you might come across in your research steps through the tree and collects the polygon data from the visible leaves into a dynamic vertex and index buffer for rendering. Unfortunately, for large polygon datasets this technique is fairly useless. The memory copying of the vertex and index data into this dynamic buffer set will kill performance. An improvement to this method maintains a static vertex buffer and collects only indices during the traversal. This is a much better design, but still not quite optimal since memory copying is a costly operation no matter what, even if you are only dealing with 16-bit indices. We will examine an alternative approach in the next lesson that does not require a dynamic buffer (although they will be handy for specific tasks as we will discover later on).

Other strategies can also be employed to speed up the frustum rejection pass of the tree by reducing the number of plane/box tests that must be performed.

One of the most popular is called *frame coherence* (or sometimes *temporal coherence*). It works by having each node remember the first frustum plane that caused the node to be rejected so that in the next frame update, we can test the failed plane first and hopefully benefit from the fact that the node is still outside the frustum. This takes advantage of the fact that between any two given frames (a small time step), a node that was invisible last time is likely to be invisible again as a result of the same plane failure. The basic idea is that the node says, “The last time I was tested, I failed against frustum plane N, so let me check plane N first this time because it is likely that in the short amount of time that has elapsed, I will fail again against this plane and get rejected straight away”. This reduces the number of plane/box tests down from 6 to 1 when all goes well.

Just as we can reject large swaths of the tree when a node is outside the frustum, another important optimization can be implemented when a node’s bounding volume is found to be contained completely *inside* the frustum. If a node is totally inside the frustum then we know for certain that all of its child nodes must also be contained inside the frustum also. Thus, once we find a node fully contained in the frustum, we no longer have to test the bounding volumes of its children -- we can traverse immediately to its leaf nodes and render them. If we do not allow for node spanning polygons (i.e., they were clipped to the nodes during compilation), hardware clipping can also be disabled since we know that none of the polygons contained within the frustum will require clipping. This is not really a major savings these days since hardware clipping is quite fast, but it is worth noting nonetheless. If you ever need to implement a software renderer, this would be something to factor in.

Finally, another common optimization to reduce plane testing involves the child nodes benefiting from information that was learned during the parent node’s frustum tests. This works for cases where there was partial intersection. We know for example that if a node is found to be inside a frustum plane, then all of its children must also be inside that frustum plane. Therefore, there is no need to test that frustum plane against any of the child nodes. We can always assume that the child currently being tested against the frustum is inside that plane without performing any test.

All of the above techniques and optimizations will be fully explained and implemented in the next lesson when we discuss rendering our spatial trees in more detail. They have been introduced only briefly here to make you aware of the complexities of such a system and the various optimizations that can be performed.

14.2.3 Hierarchical Ray Intersection

One of the most common intersection routines used in games and related applications is the intersection test of a ray with a polygon. Such techniques are used to calculate light maps, form the narrow phase of our collision detection system, and determine whether line of sight exists between two objects in the game world. For example, we often wish to determine whether a ray cast from one location in the scene to another is free from obstruction (line of sight). The only way to know that without performing spatial partitioning is to intersection test the ray with every polygon in the scene. Only when we reach the end of the polygon list with no intersections found do we know that the ray is free from obstruction and a clear line of sight exists. Unfortunately, if our scene is comprised of many thousands of polygons, that query is going to be unsuitable for real-time applications. Furthermore, we may often want to perform such tests many times in a single frame update. Spatial partitioning speeds up such ray/scene intersection tests by reducing the per-polygon component. The only polygons that need to be individually tested against the ray are the ones inside the leaf nodes whose bounding boxes are intersected by the ray. To demonstrate this, in this next example we will assume that a scene has been spatially divided into a tree, resulting in 16 leaf nodes.

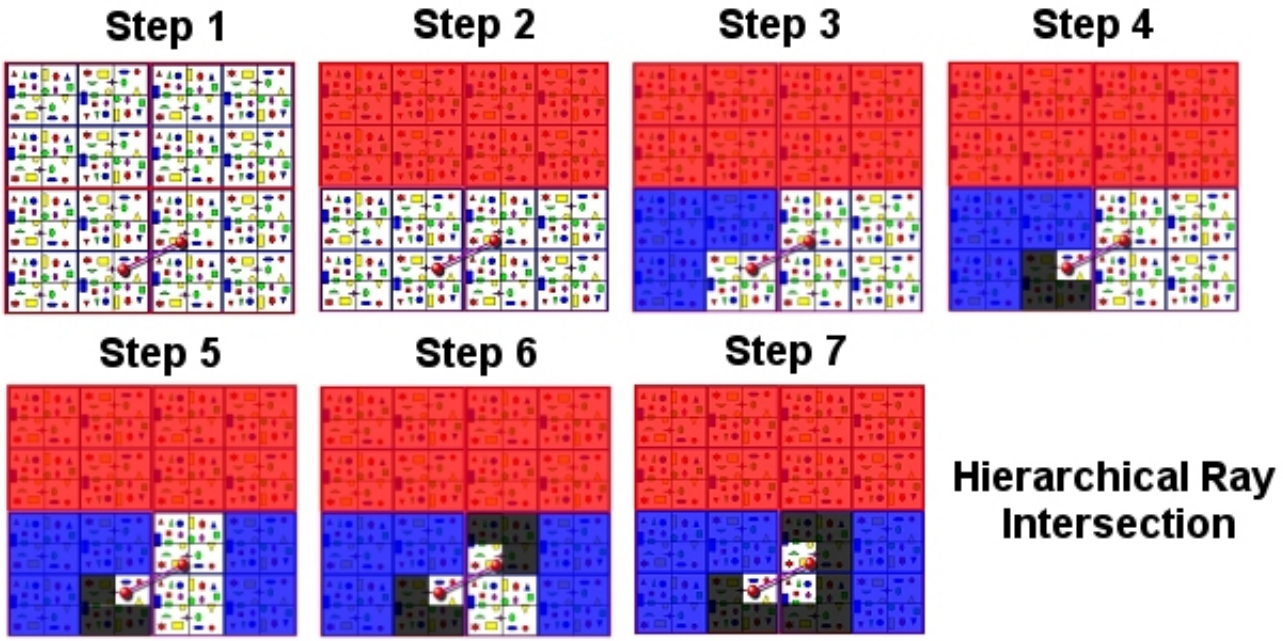


Figure 14.21

In Step 1 we see the ray end points (the two red spheres) and a purple line joining them. We can also see that the scene contains many polygons.

To find the polygons our ray intersects, we need to feed the ray through the tree and find which leaf nodes it eventually pops out in. We would start by feeding in the ray to the root node and testing it against the bounding volumes of each child of the root. As shown in Step 2, in this first stage, two of the root's children would be rejected because the ray does not intersect them. The rejected quadrants are highlighted in red.

Since we have found children of the root whose bounding volumes are intersected by the ray (the bottom two quadrants of the scene) we must send our ray into each child node. We will follow the path of the bottom left child node first. In Step 3 we can see that once in the bottom left quadrant of the root, we would then test the ray against the four child quadrants of this node. The ray is only found to be intersecting one of its children, so three are rejected from further consideration. These are highlighted in blue in the image.

From the blue children in the bottom left quadrant, only its bottom right quadrant node was intersecting the ray so we send it down to that node (Step 3). In Step 4 we show our ray visiting this node and being tested against that node's four children. Of its children, only one of them intersects the ray and three are rejected. The three rejected children are highlighted black in Step 4. At this point we send the ray into that child and find it is a leaf node which contains polygon data. We add the polygon data to a container so that we will have access to it when traversal is complete.

We have now traversed to the bottom of the tree entered via the bottom left child node of the root, so now it is time to send the ray down the root node's bottom right child. In Step 5 we see that as we visit the bottom right child of the root and test against its four children, two of them are not intersecting the ray (the blue ones). Two of the children are however, so we must traverse with our ray into each. First we start with the top left child where the ray is tested against its four child nodes and found to be contained in only one (Step 6). This one is also a leaf node, so the polygons are added to the polygon container. The rejected children are highlighted black. We have now finished with that branch of the tree so we step up a level and visit the other node our ray intersected -- the bottom left child of the bottom right child of the root (Step 7). Once again, we pass our ray into this node and test against its four children and find our ray is only in one of them. The ones that are rejected are highlighted in black. At this point we have reached the bottom of the tree, so we return. The recursive process unwinds right up to the root node and we have a container with polygons that were contained in the three intersected leaf nodes. These polygons can then be tested one at a time to see if an intersection really does occur.

In this simple example we have used a scene divided up into 16 leaf nodes so only $13/16^{th}$ of our polygon data would have to be tested at the per-polygon level. However, in a real situation we would have the scene divided up into many more leaves and the polygon data collected would be a mere fraction of the overall polygon count of the scene. To prove its efficiency, we can see that by performing four ray/box intersections tests at the root node we immediately reject half the total number of polygons in the scene regardless of the size of that scene.

Remember that although the spatial partitioning examples given here are partitioning polygon soups and storing individual polygons at the leaf nodes, this need not be the case. If your scene is represented as series of meshes for example, you could modify the node structure to contain a linked list of meshes instead of an array of polygons. Assigning a mesh to the tree would involve nothing more than sending its bounding volume down the tree and storing its pointer in the leaf nodes its bounding volume ends up

intersecting. You will often have a single mesh assigned to multiple leaf nodes, so you will have to make sure you do not render it twice during your render pass. In Lab Project 14.1 we will actually implement a spatial tree that manages both static polygon soups and dynamic meshes/actors. The static world space geometry loaded from the IWF file will be (optionally clipped and) assigned to the leaf node's polygon array. Each dynamic object (actor) will contain a list of leaves in which it is contained. When a dynamic object's position is updated, we will send its bounding volume through the tree and get back the list of leaves it intersected. We will store these leaf indices in the object structure. Before rendering any object, we will instruct the spatial tree to build a list of visible leaves. The object will ask the tree whether any of the leaves it is contained in are visible and if so, we can render the object. To be sure, there are other ways to manage this relationship, and we will talk more about it in the next lesson. For now, this gives you a fairly high level view of what is to come.

Now that we basically understand what spatial partitioning is, we will now look at some of the more common choices of spatial partitioning data structures (trees) that are used in commercial game development. There are many different types of trees used to spatially partition scenes into a hierarchy and the tree type you use for your application will very much depend on the scene itself and the type of partitioning it requires. In this lesson and the next we will discuss and implement quad-trees, oct-trees, and kD-trees. In Chapters 16 and 17 we will examine BSP trees. Keep in mind that there are a variety of different ways that each of these trees can be implemented. In this chapter we will discuss the vanilla approaches that tend to be the most common for each tree type. We will begin our exploration with quad-trees since we have already laid some foundation in the prior section.

14.3 Quad-Tree Theory

A quad-tree is essentially a two dimensional spatial partitioning data structure. This does not mean that it cannot be used to spatially partition three dimensional worlds, only that it spatially subdivides the world into bounding volumes along the (typically) XZ axis of the world. In a vanilla implementation where polygon data is only assigned to the leaf nodes, each one of these leaves contains all the geometry that falls within the X and Z extents of its bounding volume. No spatial subdivision is done along the third axis (traditionally, the Y axis). That is, the Y extents of every leaf node will be the same; the maximum and minimum extents of the entire scene, which creates bounding volumes with identical heights. As discussed in the last section, each node in the tree has four children that uniformly divide its space into quadrants.

Because of the two dimensional subdivision scheme used by the quad-tree, it is ideally suited for partitioning scenes that do not contain many polygons spread over a wide range of altitudes. For terrain partitioning, a quad-tree is a logical choice since a terrain's polygon data is usually aligned with the XZ plane. During a frustum culling pass, there will not be many times when parts of the terrain that are not visible will be situated above or below the frustum. Because every leaf node shares the same height, we will never be frustum culling geometry that is above or below the camera in a given location on the terrain. We might say that the quad-tree allows us to frustum cull data that is either in front or behind us and to the left or right of us. It is very unlikely that you will be standing on a terrain square and have parts of the terrain located nearby but very far above you or below you. Of course, there may be times

when you might be standing at the foot of a hill and perhaps the top of that hill could be culled because it is situated above the top frustum plane, but in general quad-trees work fairly well for terrains.

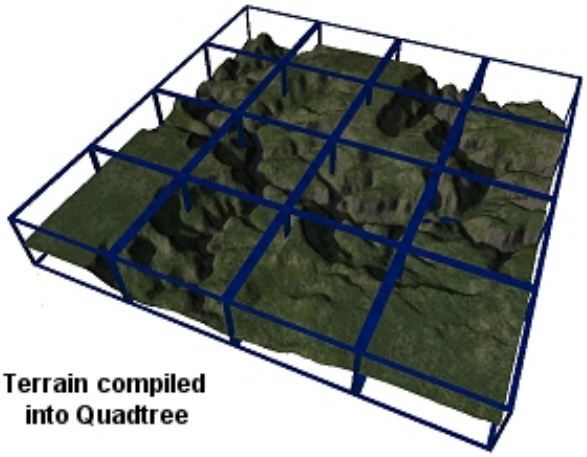


Figure 14.22

As you can see in Figure 14.22, a terrain like this has much larger dimensions along the X and Z axes of the world, so little would be gained from subdividing space vertically as well. This would introduce many more nodes into the tree and make traversals slower for very little (if any) benefit.

Figure 14.22 shows a terrain compiled into a quad-tree and although in reality this terrain would be compiled into many more leaf nodes than illustrated here, it should be obvious why this would be an efficient partitioning scheme for a terrain.

The height of each of the leaf node's bounding volumes can either be the max Y extents of the polygons that exist in that leaf node or it can be taken from the Y extents of the bounding box compiled for the entire scene at the root node. In the previous examples and in Figure 14.22, we have used the Y extents of the bounding box compiled for the entire scene for each leaf node's Y extents. We can see in Figure 14.22 that this generates nodes of identical height throughout the entire tree. In certain cases, leaf nodes may be more efficiently frustum culled or rejected from polygon queries earlier if the Y extents of each node's bounding box is not inherited from the bounding volume for entire scene but is instead calculated at node creation time from the polygon data in that node. Calculating the Y extents for a node would be easy -- we could just loop through each polygon that made it into that node during the compile and record the maximum and minimum Y coordinates of the polygon set. This will yield the minimum and maximum Y coordinates of any polygons stored at that node (or below it) which can then be used as the Y extents of its bounding box. This type of quad-tree (which we will refer to as a *Y-variant quad-tree*) generates children at each node which may not fill the space of the entire parent node, but will still always be totally contained inside it.

Figure 14.23 shows a how a quad-tree node normally has its space uniformly subdivided for each of its children. The height of all child nodes is equal to the height of the parent, even if there is no polygon data stored at that height. The circular inset shows how the children of the quad-tree node might look if, when each child node is created, the Y extents of its bounding volume are calculated from the polygon data that made it into that node. The Y-variant quad-tree is a favourite of ours here at the Game Institute as it performs consistently well in all of our benchmarks.

Before we discuss how to create a quad-tree, we should look at situations where it might not be the best choice of spatial manager due to its two dimensional spatial partitioning. A space-combat scene is one such scenario where the quad-tree might not be the best choice.

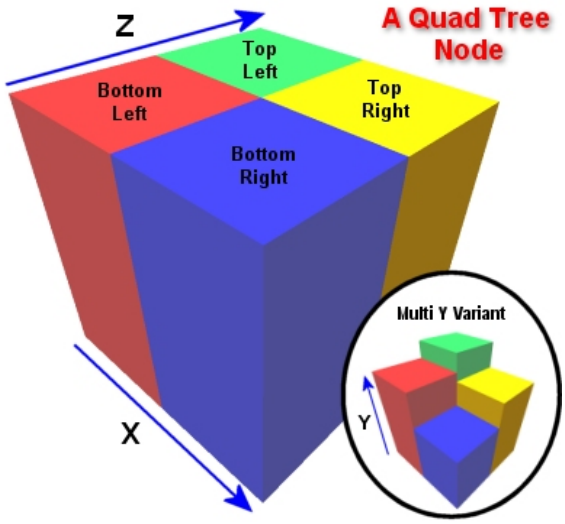


Figure 14.23

In such a scene, you could (for example) have a hundred complex spaceship meshes all situated at exactly the same X and Z coordinates in space but at different positions along the Y axis of the coordinate system. We can think of these spaceship models as being positioned in the world such that they would look like they were on top of each other when the scene was viewed from above. If we were using a quad-tree to spatially manage this scene of dynamic objects, all one hundred space ship models would exist in a single leaf node. When the camera entered this leaf node, all of the spaceships in this node would be rendered. This would be true even if all the ships were positioned far above and below the frustum. Essentially, we would be sending one hundred space ship models through the transformation pipeline even if none of them can be seen. Therefore in this example, where the scene has large Y extents, we would rather use a spatial partitioning technique that allows for the scene to be subdivided vertically as well. Although the examples given are associated with efficient frustum culling, the same is true for collision queries. If our swept sphere's bounding volume intersected that same leaf node, the polygons of all one hundred space ships would be collected by the broad phase and sent to the narrow phase as potential colliders (assuming we did no further bounding volume tests). However, the space ships could be positioned nowhere near the swept sphere.

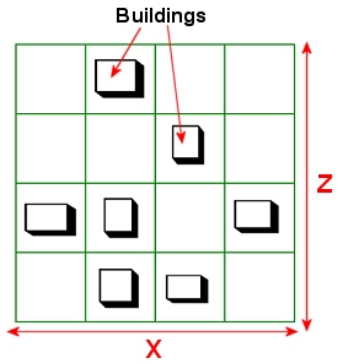


Figure 14.24

Another example where a quad-tree might not be the best choice is when representing a scene that models a cityscape with towering skyscrapers. We will use a very simple example to demonstrate why this is the case.

In the Figure 14.24 we see a scene consisting of seven tall buildings compiled into a quad-tree. In this first image we are looking at the scene from the top down perspective and at first glance this scene appears to fit nicely into a quad-tree.

In this example the scene has been divided into 16 leaf nodes along the X and Z axes of the world. What cannot be seen from this two dimensional representation is how high each building is, and thus how tall the bounding boxes are. In Figure 14.25 we see another view of the scene rendered three dimensionally to better demonstrate this point.

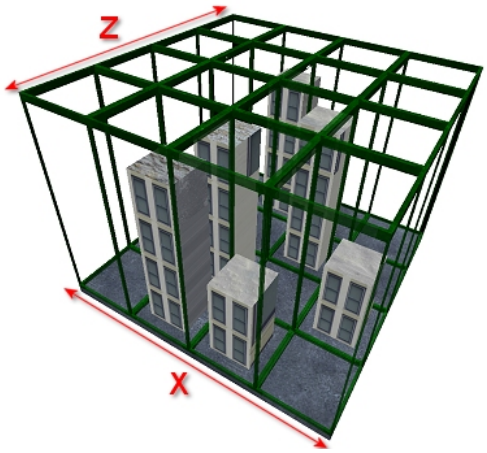


Figure 14.25

In this example, the Y extents of each quad-tree node are the same as the Y extents of the entire scene's bounding box, generating leaf nodes of identical size along the Y axis. Now let us imagine that the camera is located in front of the first two buildings at ground level with a rather narrow frustum.

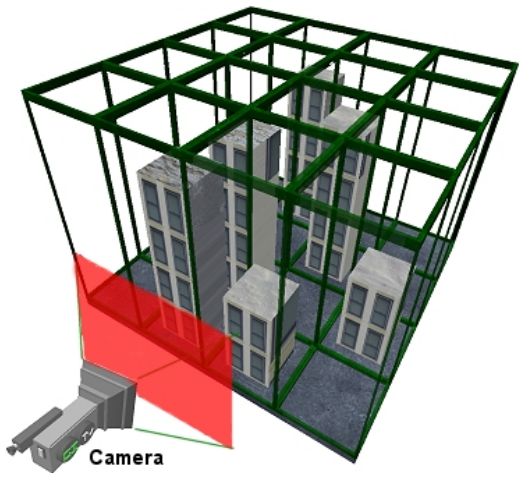


Figure 14.26

The red highlighted area in Figure 14.26 illustrates the section of the two front buildings that are actually inside the frustum and would need to be rendered. The camera can only see the bottom section of the two front buildings and therefore it is only the polygons comprising the bottom sections of each building that need to be rendered. However, because the polygons in each building are assigned to a single leaf node, when the leaf node is partially visible all of the polygons in that leaf are rendered. In this image we can see that the front three leaf nodes (with respect to the camera) can be partially seen from the camera position and as two of those leaf nodes contain buildings, these building will be rendered in their entirety. This is a shame as there are many polygons in each building which are situated well above the range of the frustum and they would be needlessly transformed and rendered (or collected as potential colliders). If you imagine a large cityscape scene consisting of hundreds of tall skyscrapers and imagine that the camera is usually situated at the ground level, using a quad-tree to partition the scene would result in many polygons in the upper regions of these buildings being rendered when they cannot even be seen. An oct-tree (discussed later) might prove to be a better choice of partitioning scheme in this instance.

You may be thinking that the simple answer is to always use an oct-tree instead of a quad-tree when working in three dimensions, but that is not a wise rule to live by. In our terrain example, nothing would be gained (or very little) by spatially subdividing the scene vertically. All we would achieve is a larger tree that is slower to traverse. More AABB/Frustum tests would need to be performed to reject a section of terrain that, in a quad-tree, would fit in a single leaf node and be rejected with a single AABB/Frustum test.

Note: Do not assume that an oct-tree is the best solution even if your scene does contain a large distribution of objects along the Y axis. Oct-trees create many more nodes which makes them slower to traverse and causes the polygons to be rendered in smaller batches. On modern hardware this might cause it to underperform with respect to the quad-tree. In most of our tests performed with the partitioning and rendering of static polygon data, the Y variant quad-tree outperformed the oct-tree in almost every case. However, this might not be true for a mesh based tree and would certainly not be true for our collision system's broad phase. We would not want to send polygon data to the expensive narrow phase which is above or below our swept sphere's volume. However, for rendering purposes, the quad-tree has been consistently hardware friendly. The lesson is to always to test the tree types on a variety of different scenes and machine configurations before deciding which one to use. Of course, you do not have to use the same tree for your collision geometry as you do for your render geometry. For example, you could compile your collision data using an oct-tree but render the scene data using a quad-tree.

14.3.1 Partitioning a Quad-Tree Node

When building a quad-tree node, we will essentially have a list of polygons that are contained in the parent node volume that need to be assigned to its four child nodes. The parent volume will be subdivided into four equally sized volumes along the X and Z axes. The centerpoint of the parent node's bounding box describes the position where the edges of each of the four child node's bounding boxes will meet. Each child node is created and is assigned a bounding volume representing a quadrant of the parent node (top left, top right, bottom left or bottom right). Once we have the child nodes and their bounding volumes calculated, we can start to test which child node(s) a given polygon in the parent node's polygon list is in. It is possible that a polygon in the list passed down from the parent might span the bounding volumes of multiple child nodes. In such a case we can either choose to add the polygon to the polygon list of each child it intersects, or we can clip the polygon to the bounding volumes and store each polygon fragment in the child for whose bounding volume it will now be totally contained.

Placing the polygon in multiple leaf nodes does come with its fair share of problems. We will need to make sure that we do not render or query the polygon multiple times if more than one of the leaves in which it exists is being rendered or queried. This can add some traversal overhead, although not much. Another problem with not clipping the polygons is that when a leaf node is rendered or queried, the node no longer describes the polygons exactly stored in that volume. For example, we might assign a polygon to a leaf node that is much larger than the node. When that node is visible, we have a looser fit for the exact data that can be seen and should be processed. Of course, the problem with clipping is that we essentially perform a process that creates two polygons from one. If this happens hundreds of times during the building procedure it is not uncommon for the polygon count to grow between 50 to 90 percent depending on the type of tree you are using and the size of the leaf nodes. The larger the leaf nodes, the less clipping will occur but more polygons will be contained in a single leaf.

In our implementation, we will provide options for clipping the static polygon data to the bounding volumes of the tree and will also implement a system that elegantly handles the sharing of polygons between multiple nodes when clipping is not being used.

We will store in our node its bounding volume and two clip planes that are used for the classification and clipping of polygon data during the tree building process. We will discuss how to clip polygons to planes later in this lesson. The benefit of clipping is that any node in the tree will contain an exact fit of

the data contained inside it and more importantly, no polygon will ever belong to multiple leaf nodes. Of course, it does mean that whenever a split happens we introduce more polygons into the scene.

> **Note:** Until otherwise stated, we will assume for the sake of the next discussion that we are clipping our polygon data to the nodes in which they belong. Later we will discuss the non-clip option.

Please note that in this section we are currently talking about the assigning of static world space polygon data to the tree, such as the internal geometry loaded from a GILES™ created IWF file. As this data never changes throughout the life of the application, we will clip it exactly to the tree at compile time and store the polygon data in the leaves. Later we will discuss how we can also link dynamic objects to leaf nodes in the tree. These objects are not clipped and are linked to the leaf nodes implicitly by storing a list of leaves they are contained within. Meshes and actors do not have their polygon data stored in the tree. Instead we will maintain an internal array of leaf indices in which the object currently resides. The spatial tree will expose a method that will allow dynamic objects to pass their bounding volume down the tree and get back a list of leaf indices in which they are currently contained. The dynamic object can optionally store these indices and only render itself if any of these leaves are visible (although the system will maintain this information internally for later querying if desired).

Because a given node is really just an axis aligned bounding box, generating the clip planes for a quad-tree node is delightfully easy. Two planes will be needed: one that will split the node's volume halfway along its depth (Z) axis and another that will split the volume half way along its width (X) axis. We know that the normal to a plane that will divide its depth in two will be <0, 0, 1> (or <0, 0, -1> as it represents the same clip plane) and that the centerpoint of the node's bounding volume will be a point on that plane as shown in Figure 14.27.

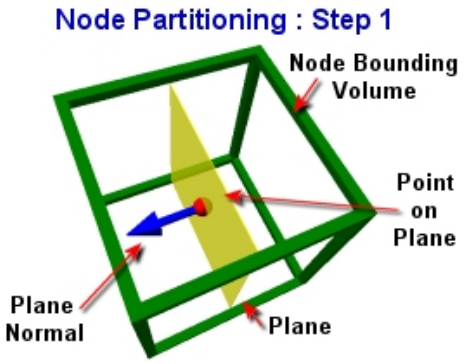


Figure 14.27

In this image we are assuming that the blue arrow is the plane normal pointing along the world Z axis and that the red sphere is the centerpoint of the node's bounding volume. Thus, these two pieces of information are all we need to describe the clip plane shown as the yellow slab in Figure 14.27.

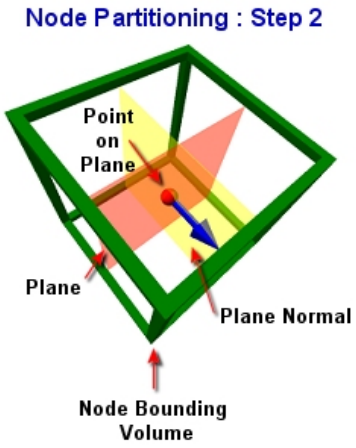


Figure 14.28

The creation of the second clip plane is equally as easy. Once again, the centerpoint of the bounding volume describes the point on the plane and this time we use a normal of <1, 0, 0> so that we have a plane that divides the volume's width in two.

Once we have generated the two planes, we take each polygon in the node's list and clip it to each plane. This is done in two passes. For each polygon in the list we classify it against the first node plane. If it is either in front or behind the node we leave it in the list and continue on to the next polygon. If we find that a polygon is spanning the plane we split it at the plane. The original polygon in the list is deleted and the two new split polygons are added to the list. After every polygon in the list for this node has been tested against the first plane, any polygon that was spanning the plane will have been removed from the list and replaced with two polygons that fit entirely in the front and back half space of the node's volume.

We then clip this list against the second plane using the same scheme. That is, each polygon is classified against the plane and if not spanning, the second plane is left in the list unaltered. If it does span the second plane, we split the polygon in two, deleting the original from the list and adding the two split fragments in its place. At the end of this process, we will have not yet determined which child nodes each polygon should be assigned to, but we know the polygons have been clipped such that every polygon will neatly fit into exactly one child node (one quadrant).

The following code shows this step. Do not worry too much about how the actual functions that are called work at the moment; we will get to all that later when we implement everything. For now just know that this code would be executed during the building of a node. The node is passed an STL vector of all the polygons that have made it into that node during the compilation process so far. For the root node, this vector will contain all the polygons in the scene. Each polygon is assumed to be represented by a CPolygon class that has a method called 'Classify' which returns a flag describing whether it is in front, behind, or spanning a plane. It also has a method called 'Split' which splits the polygon to a plane and returns two new CPolygon structures containing the split polygon fragments. Remember, this code is not yet trying to determine which polygon in the list should be passed to which child node, it is just clipping any polygons that straddle the quadrant borders.

The first thing we do is generate the two clip planes using the D3DXPlaneFromPointNormal method. This method accepts as its parameters a point on the plane and a normal and returns (via the first parameter) the plane represented as a D3DXPlane structure (in a,b,c,d format). For both planes, the point on plane is simply the centerpoint of the node's bounding box (2nd parameter). For the plane normal of the first plane we pass in the vector (0,0,1) which is the world Z axis; for the second plane we pass the vector (1,0,0) which is the world X axis.

```
D3DXPlane ClipPlanes[2];

D3DXPlaneFromPointNormal( &ClipPlanes[0], &((BoundsMin + BoundsMax) / 2.0f),
                          &D3DXVECTOR3( 0.0f, 0.0f, 1.0f ) );

D3DXPlaneFromPointNormal(  &ClipPlanes[1], &((BoundsMin + BoundsMax) / 2.0f),
                           &D3DXVECTOR3( 1.0f, 0.0f, 0.0f ) );
```

Notice in the above code how we have allocated an array of two D3DXPlane structures on the stack which will receive the clip planes that we calculate. In the first element we store the XY plane and in the second element we store the YZ plane.

Now that we have both planes temporarily calculated for the current node, we will classify the polygons in the list against each one. Therefore, we set up an outer loop for each plane and an inner loop that tests each polygon against the current plane being processed.

```
    // Split all polygons against both planes
    for ( i = 0; i < 2; ++i )
    {
        for(PolyIterator= PolyList.begin();PolyIterator != PolyList.end(); ++PolyIterator)
        {
            // Store current poly
            CurrentPoly = *PolyIterator;
            if ( !CurrentPoly ) continue;
```

The first thing we do is test the current polygon being processed to see if its pointer is NULL, and if so, we skip it. You will see in a moment why we perform this test.

Now that we have a pointer to the current polygon we want to test we will call its Classify method and pass in the current plane. The Classify method of CPolygon simply returns a value that describes the position of the polygon with respect to the plane. The four values it can return are CLASSIFY_INFRONT, CLASSIFY_BEHIND, CLASSIFY_ONPLANE and CLASSIFY_SPANNING. As you can see, these tell us whether the polygon we are testing is in front or behind the plane or whether it is spanning the plane. In this loop, we are looking for polygons that are spanning the current plane being tested because if we find a polygon spanning a plane, we must spilt it into two new polygons at that plane. Here is the remainder of the code.

```
            // Classify the poly against the first plane
            ULONG Location = CurrentPoly->Classify( ClipPlanes[i] );

            if ( Location == CPolygon::CLASSIFY_SPANNING )
            {
              // Split the current poly against the plane,
              // delete it and set it to NULL in the list
                CurrentPoly->Split( ClipPlanes[i], &FrontSplit, &BackSplit );

                delete CurrentPoly;
                *PolyIterator = NULL;

                // Add these to the end of the current poly list
                PolyList.push_back( FrontSplit );
                PolyList.push_back( BackSplit );

            } // End if Spanning

        } // Next Polygon

    } // Next Plane
```

As you can see, if the classification of the current polygon does not return CLASSIFY_SPANNING then we leave it in the list and skip on to the next one. That does not mean of course that this same polygon will not be clipped when the second plane is tested in the second iteration of the outer loop. If

the polygon is spanning the current plane then we call its Split routine. This function takes three parameters. The first is the plane we would like to clip the polygon against and the second and third parameters are where we pass in the address of CPolygon pointers which on function return will point to the new split fragments.

After this function returns, FrontSplit and BackSplit will contain the two polygon fragments that lay in front of the plane and behind it, respectively. At this point we no longer want the original polygon in the list as it will now be replaced by these two fragments. Therefore, we delete the original CPolygon structure and set that element in the STL vector to NULL. We then add the front and back splits to the polygon list. Of course, these split fragments may each get clipped again when they are tested against the second plane in the second iteration of the outer loop.

Notice in the above code that whenever we split a polygon, that original polygon is deleted and its pointer in the STL vector is replaced with a NULL. The two split polygons are added to the end of the vector. Now you can see why we did the polygon pointer test for NULL at the top of the list. When processing the second clip plane, many of the pointers in the list may be NULL. There will be a NULL in the list for every original polygon that was split by the first plane.

Now that we have the polygons such that every polygon is contained in exactly one quadrant of the node, it is now time to build four children polygon lists. That is, we need to calculate which polygons will need to be passed into each child node. We will build a list for each quadrant which will result in four polygon lists which we can then pass into the child nodes during the recursive build process. The next section of code shows these four polygon lists (STL vectors) being compiled.

```
PolygonList            ChildList[4];

// Classify the polygons and sort them into the four child lists.
for(PolyIterator = PolyList.begin(); PolyIterator!=PolyList.end(); ++PolyIterator )
{
 // Store current poly
 CurrentPoly = *PolyIterator;
 if ( !CurrentPoly ) continue;

 // Classify the poly against the planes
 Location0 = CurrentPoly->Classify( ClipPlanes[0] );
 Location1 = CurrentPoly->Classify( ClipPlanes[1] );

 // Position relative to XY plane
 if ( Location0 == CPolygon::CLASSIFY_BEHIND )
 {
    // Position relative to ZY Plane
    if ( Location1 == CPolygon::CLASSIFY_BEHIND )
         ChildList[0].push_back( CurrentPoly );
    else
         ChildList[1].push_back( CurrentPoly );

  } // End if behind

 else
  {
    // Position relative to ZY Plane
```

```
    if ( Location1 == CPolygon::CLASSIFY_BEHIND )
         ChildList[2].push_back( CurrentPoly );
    else
         ChildList[3].push_back( CurrentPoly );

  } // End if in-front or on-plane

} // Next Polygon
```

The code first allocates four empty CPolygon vectors that will contain the polygon lists for each child node that we generate. We then loop through each polygon in the clipped list that we just modified. Remembering to skip past any polygon pointer that is set to NULL in the array, we then classify the polygon against both of the node's clip planes and store the results in the Location0 and Location1 local variables. Finding out which list the polygon should be assigned to is now a simple case of testing these results.

We can see in the above code that if the polygon is found to be behind the first clip plane (the Z axis split) then the polygon is obviously contained in the back half space. We then test to see what its classification against the second clip plane was. Remember the second clip plane is the plane that splits the width of the volume and has a normal (1,0,0). If it is behind then we know that not only is the polygon in the back halfspace of the node, but it is also in the left halfspace behind the first clip plane. Thus, this polygon must be contained in the top left quadrant. The else case says that if we are not behind the second clip plane but we are behind the first clip plane, then this polygon must be contained in the top right quadrant of the node. We perform the same tests when the polygon is found to be in front of the first clip plane to determine whether it belongs to the bottom left or bottom right quadrant.

As the above code shows, once we find the quadrant a polygon is in, we add it to the relevant polygon list. After this code has executed, ChildList[0] will contain the polygons for the top left quadrant and ChildList[1] will contain the polygons for the top right. ChildList[2] and ChildList[3] will contain the polygons for the bottom left and bottom right quadrants, respectively.

At this point, all that would be left to do is allocate the four child nodes, construct their bounding boxes as quadrants of the parent node's volume and recur into each child sending the list that was compiled for it by the parent node and the whole process repeats. Only when the list of polygons is small or the bounding box of the child node is small do we decide to stop subdividing and just assign the polygon list to the node, making it a leaf.

> **Note**: Do not worry if the above code did not provide enough insight into how to fully create a quad-tree. It was intended only to show the clipping that happens at each node during the build process. Later in this lesson we will walk through the code to a quad-tree compiler line by line.

Notice that the clip planes will be generated only during the construction of the node's child lists and they will not be stored. The only thing we store in the node (apart from any polygons) is its axis aligned bounding box and the child node pointers. Of course, you could decide the store the clip planes in the node as well if you think you will need them at a later stage. It is possible that you may wish to perform some query routines on the tree using the clip planes instead of the axis aligned bounding boxes, but we do not in our lab project. However, we will store the clip planes in the nodes of the kD-tree that we implement (and in the BSP tree, as we will see later in the course).

We now have a good idea of exactly what a quad-tree is, and this will go along way towards our understanding of the other spatial tree types. Later in this chapter we will cover the source code to a quad-tree compiler which we can then use in our applications. Each tree type we develop will all be derived from an abstract base interface, so we will be able to plug any of them in as the broad phase of our collision system.

That concludes our coverage of the quad-tree from a theoretical perspective, so we will now go on to discuss the other tree types. After we have discussed the various tree types, we will examine the separate processes involved in building them, such as the clipping of polygons and the mending of T-junctions introduced in the clipping phase.

14.4 Oct-Tree Theory

The great thing about the topic of hierarchical spatial partitioning is that whether we are implementing a quad-tree, an oct-tree or a kD-tree, the building and traversal of those trees are almost identical in every case. They are all trees consisting of nodes which themselves have child nodes which can be queried for intersection and traversed recursively. This is especially true in the case of an oct-tree where the only real difference between the quad-tree and the oct-tree is in the number of children spawned from each node.

In a quad-tree, each node's bounding volume is divided into quadrants along the X and Z axes and assigned to each of its four children. In an oct-tree, a node's bounding volume is divided along the X and Z axes and also along the Y axis. Therefore, each non-leaf node has its bounding volume divided into octants ($1/8^{th}$) and each non-leaf node of an oct-tree has eight children instead of four. The bounding volume of each child node represents $1/8^{th}$ of the parent node's bounding volume. Figure 14.30 shows how a single node's bounding box is divided into octants within an oct-tree, compared to being divided into quadrants for a quad-tree.

Figure 14.29

Figure 14.29 shows how the bounding volume of a non-terminal quad-tree node is subdivided into four smaller bounding volumes along its X and Z axes. In this image the child nodes each have the same height taken from the Y extents of the entire scene (the root node's bounding box). The circular inset in the image shows a variation of the quad-tree where the Y extents of each child node are calculated using the actual polygon data that was passed into that node during the building process.

In Figure 14.30 we see how the bounding volume of a non-terminal oct-tree node is divided into octants. Not only do we divide the parent node's bounding volume in two along the X and Z axes as we do with the quad-tree, but we also divide the parent volume into two along the Y axis. This creates eight child bounding volumes instead of four. The subdivision of the node's bounding volume resembles a double decked version of the quad-tree subdivision. Instead of just having Top Left, Top Right, Bottom Left and Bottom Right child nodes, the child nodes can now be described by prefixing the label with the deck to which they belong: Upper Top Left, Upper Top Right, Upper Bottom Left, Upper Bottom Right, Lower Top Left, Lower Top Right, Lower Bottom Left, and Lower Bottom Right as labeled in the diagram

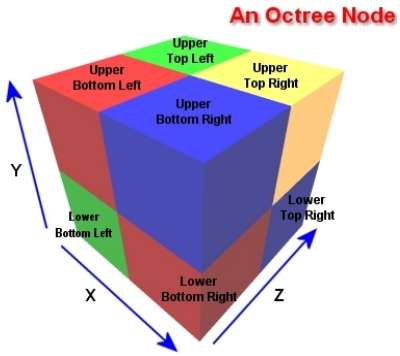


Figure 14.30

As you might imagine, building an oct-tree is a nearly identical process to building a quad-tree. The exception being of course that each non-leaf node has eight children instead of four so we have to divide our polygons into eight bounding boxes instead of four when building each node. Everything else is as before. The leaf nodes are the nodes at the end of a branch of the tree and in our implementation, are the only nodes that can contain geometry data.

The process of building an oct-tree requires only small changes to the code we saw earlier. The node structure used for an oct-tree will have pointers to eight child nodes instead of four and when building the node itself (in our implementation) we will now use three clip planes instead of two to build the clipped polygon lists for each child (see Figure 14.31).

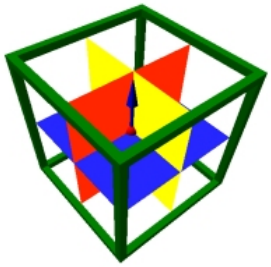


Figure 14.31

Trying to draw an image of what an oct-tree looks like in memory on a piece of paper is virtually impossible if the oct-tree is more than a few levels deep. Each node has eight children, each of which have eight children of their own, and so on right down the tree. In fact, if we think about how many leaves a 10 level oct-tree would partition our scene into, we would get a staggering result of 8^10 = 1,073,741,824 nodes at the lowest level of the tree. Not surprisingly, it is pretty much never the case that we will compile an oct-tree that has anywhere near this many levels. The good thing about oct-trees then is that they are typically going to be fairly shallow tree structures. Of course, when traversing the tree, we have eight child tests to perform at each node (instead of just four in the quad-tree case) to find the child nodes we wish to step into.

Figure 14.32 depicts a partial oct-tree that is three levels deep. The gray node represents the root node of the tree, the red nodes represented the eight immediate child nodes of the root, and if the width of a printed page was not an issue, each one of these nodes would have their own eight blue child nodes which in this example are leaf nodes. So that we can fit the image on a piece of paper we have only shown the child nodes of two of the red nodes. Remember, each one of these red nodes would have their own eight blue leaf nodes. Hopefully, this diagram will illustrate just how much the quad-tree and the

oct-tree are alike at their core. The oct-tree has double the number of branches leaving each node obviously, but it has the same arrangement with leaf nodes at the branch tips.

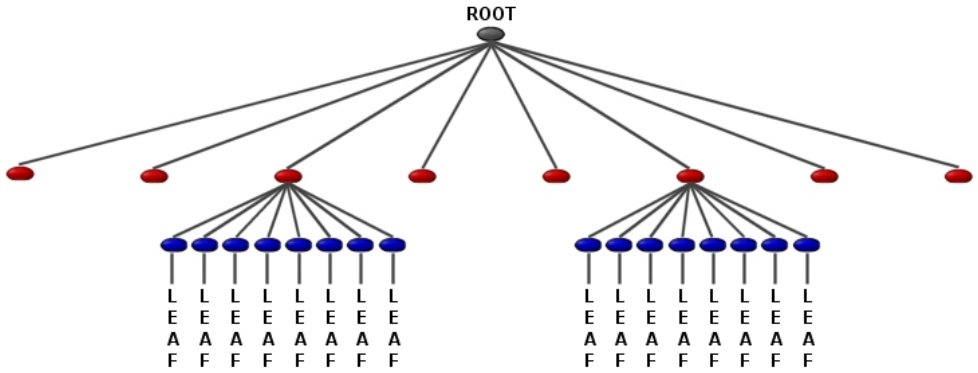


Figure 14.32

Rendering the oct-tree would also not be that different from the quad-tree case. The only real difference would be that when traversing the nodes of an oct-tree, we now have to perform Frustum/AABB tests against eight bounding volumes at each node instead of four, before we traverse down into the visible children.

As you can see from examining Figure 14.33, with an oct-tree, the scene is also divided vertically as well. Looking at the same position and orientation of the camera in this image, we can see that only the bottom two leaf nodes fall within the camera's frustum so only the polygons assigned to those leaf nodes would need to be rendered, which in this case would be the polygons in the bottom sections of each building only. The only time the top sections of the buildings would be rendered is when the camera is rotated upwards (like a person looking up at the sky) because only then would the upper leaf nodes intersect the frustum. At this point however, there is a good chance that the bottom leaf nodes of the building would no longer be inside the frustum and therefore when rendering the upper portions of the building in this example, the lower portions of each building will be frustum rejected and not rendered. Clearly this shows the advantages of an oct-tree and the limitations of the quad-tree in certain scenarios. We have seen that even if the camera is not rotated upwards, using a quad-tree, the entire building would be rendered even if the upper portions of the building cannot be seen.

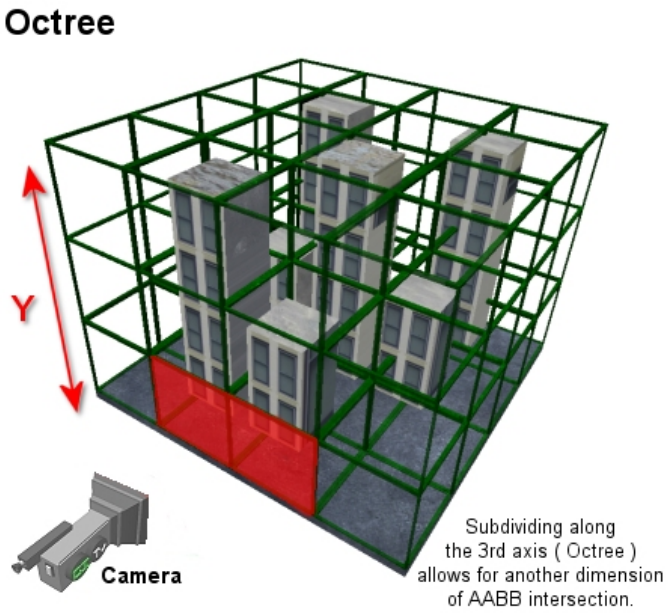


Figure 14.33

Of course, the same would also be true of intersection queries, if an object is contained in the lower leaves, only the polygons comprising the base of the buildings would need to be tested in the narrow phase. So we have discovered that the oct-tree allows us to more finely collect or cull the number of

polygons that need to be considered. If the scene geometry is distributed over a large vertical range, the oct-tree is often a better choice for multi-level indoor environments, cityscapes and areas that have immense freedom of movement along all three axes (such as a space scene). That being said, you should always benchmark your trees and find out for sure which is the best performer for a given scene. As mentioned previously, the oct-tree does suffer from creating more nodes to traverse and clips the polygon data much more aggressively. If care is not taken, you could find your polygon count doubling during oct-tree compilation.

14.5 kD-Tree Theory

A kD-tree is a partitioning technique that partitions space into two child volumes at each node. Each node contains a single split plane and pointers to two child nodes. The split planes chosen at each node are always axis aligned to the world and alternate with tree depth to carve the world into rectangular regions, much like an oct-tree. That is, at the first level of the tree, a split plane that partitions the space along the X axis might be chosen to create two child volumes. When each child is partitioned, the split plane used would divide their space along the Y axis to create two child volumes for each. For each of their children, the split plane used would be one that divides their volume along the Z axis. For their children, the process wraps around to the beginning and we start using the plane that divides space along the X axis again. This process of alternating between three axis aligned clip planes at each node repeats as we step down the tree until we wish to stop our subdivision. That node is then considered a leaf node.

Because the kD-tree partitions space in two at each node, it is a binary tree. Further, because this particular binary tree is used to partition **space** at each node, we might also refer to it as a binary space partitioning tree (BSP tree). While this is certainly true, the kD-tree is not normally what people are referring to when a BSP tree is referenced. A BSP tree is almost identical to a kD-tree except for the fact that at every node an arbitrarily oriented plane can be chosen (instead if using an axis-aligned one). This allows the BSP tree to expose very useful properties that allow for the determination of what is solid and empty space within the game world, which can then be used to optimize rendering by an order of magnitude. Because the kD-tree uses axis aligned planes, a kD-tree is often referred to as an *axis aligned BSP tree*.

The kD-tree allows space to be partitioned arbitrarily at each node as long as the clip plane is still being aligned to a world axis. That is, clip planes can exist at variable positions within the node's volume. The clip plane does not always have to divide the node's volume into two uniformly sized child volumes. This allows for the space to be subdivided such that only areas of interest (where geometry exists) get divided (more on this in a moment).

To summarize, a kD-tree is a spatial partitioning tree with the following properties.

- Each node represents a rectangular bounding volume. Just like an oct-tree or a quad-tree, the faces of each node's bounding box are aligned with the axes of the coordinate system.
- Each node stores a single split plane which is aligned to one of the coordinate system axes. This split plane does not have to cut the region into two equally child volumes (although we may wish it too).

- Each node has two child nodes representing the volumes each side of its split plane.
- The world axis used for the alignment of a node's split plane alternates with tree depth.

Building the kD-tree is much the same process as for an oct-tree or a quad-tree although there may be some differences in how the split plane at the node is chosen. To get across the basic idea of a kD-tree we will show images to illustrate the subdivision of a scene during the construction of such a tree. In this example we will simply calculate each node's split plane to cut through the center of the node's bounding volume. This will divide each node's volume into two equally sized child volumes. As you will see in the following examples, because the split plane being chosen partitions each node's volume into two along its center, we partition the scene in the same way as an oct-tree.

In this first image we show the construction of the root node. After creating the node we would loop through each of the scene's polygons and compile a bounding box that encompasses them all. This will be assigned to the root node as its bounding volume. At this point we have the root node's volume and a list of polygons that need to be passed into two child nodes. Our next task it to choose a split plane for the root. In this image we create a split plane whose normal is aligned with the world X axis (1,0,0). In this example we are using the center of the bounding volume as the point which describes the plane during plane creation and as such the plane cuts through the center of the box dividing its width in two.

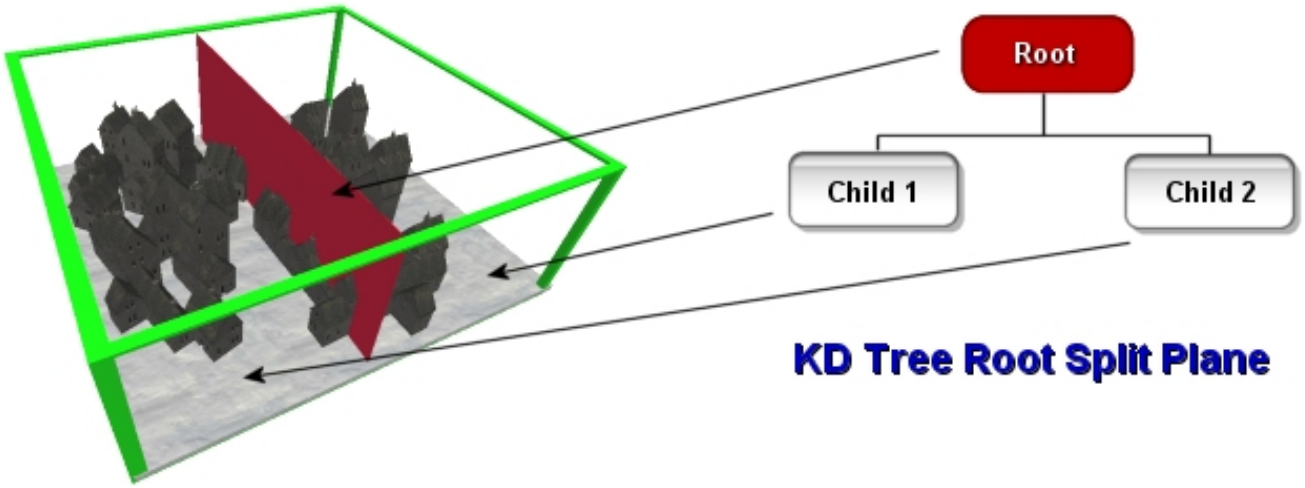


Figure 14.34

At this point we store the split plane in the node and loop through each polygon in the list and classify them against this plane. This allows us to build two polygon lists (one for each child node) describing the polygons that belong in each list. We will assume for this demonstration that we are once again clipping our geometry to the partitions, so any polygon in the root list that straddles the plane will be clipped in two by that plane and the each child fragment added to the respective child list.

At this point we have to recur into each of the children and perform exactly the same task (just as in the oct-tree/quad-tree case). The difference being now that as we step down to each level of the tree, we alternate the normal that we are going to use for the node's split plane. In Figure 14.34 we can see that at the first level in the tree we used a plane normal which is equal to the X axis of the coordinate system. This cuts the box's width in two.

In the next level of the tree we have to switch the clip plane to use one that is aligned to the Z axis of the system (0,0,1). That is, every node at the second level of the tree will use this same normal for its clip planes, as shown in Figure 14.35.

> **Note:** It does not really matter the order in which you alternate the clip planes used at each level of the tree, as long as you alternate between the three as you step through the levels of the tree. That is, you could use a plane normal equal to the world Y axis at the root, one that is equal to the world Z axis at the second level and one that is equal to the world X axis at the third level. As long as you repeat the pattern, everything will be fine and you will have a valid kD-tree.

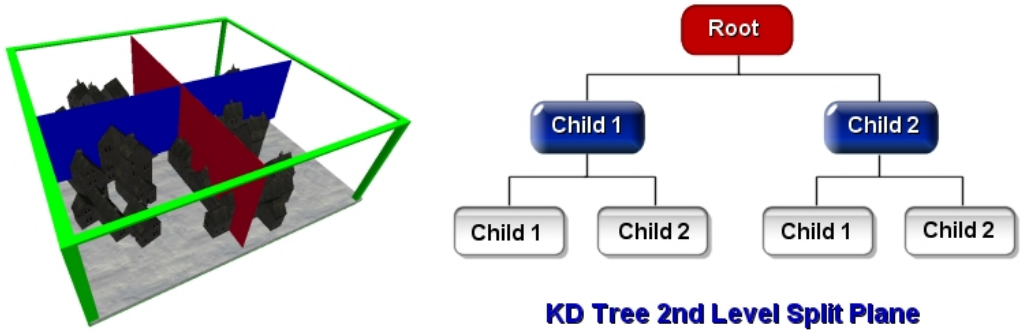


Figure 14.35

Figure 14.35 illustrates what happens when we step into each blue child of the root. For each child we are passed a list of polygons that are known to fit inside this child. We then compile a bounding box for that child based on its list of polygons and store it in the node. A split plane is then chosen which divides the child volume in two. Because we have stepped down a level, we alternate the world axis we use as our plane normal. In this example, the center point of the box is still used to construct the plane, which equates to a plane that splits each child into two equally sized children. In Figure 14.35 a plane aligned with the XY plane (the blue plane) of the coordinate system is being used for each child node, which carves each of their volumes along the depth coordinate. The polygons passed into each child node are then classified against (and clipped to) the split plane to build two child lists for each of its children (the white nodes in the hierarchy diagram).

In Figure 14.36 we see the construction of the next level of the tree, where the children of each child of the root are constructed.

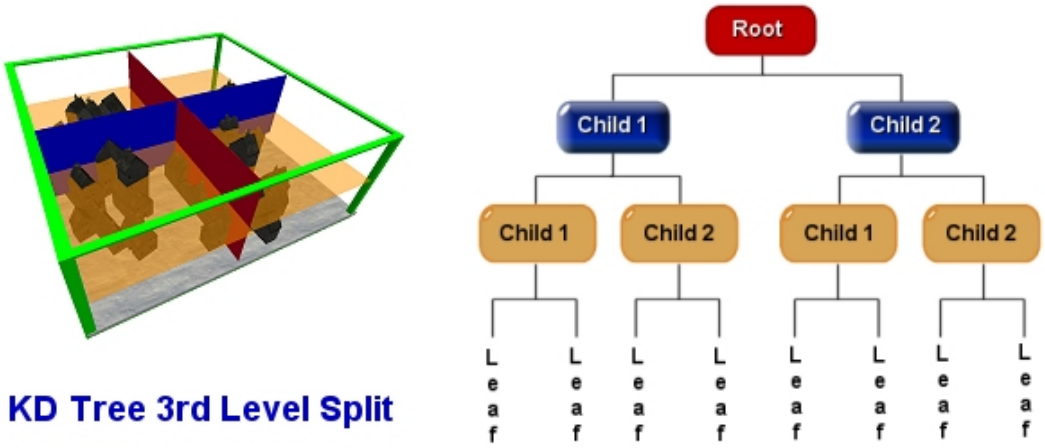


Figure 14.36

In Figure 14.36, when we step into the third level of the tree, the split plane alternates again. This time a plane normal aligned with the Y axis of the coordinate system is chosen. This creates a plane that carves each orange node in two vertically. Once again, the polygons that made it into each orange node would be classified against its plane to create two polygon lists for each of its children. In this diagram, we are assuming that when we stepped down to the fourth level of the tree, the child node's polygon list was considered to be so small that it was not worth subdividing further. At this point, we create leaf nodes which contain the actual polygon data. A leaf node is just a node like any other, with respect to representing an area of space. It has a bounding volume, no children, and it has polygon or mesh data associated with it.

We will see later that building and querying a kD-tree is even easier than both the oct-tree and the quad-tree since we have only one plane and two children to worry about at each node and we are simply trying to determine in which of the two children our query volume belongs. Performing a ray intersection test on a kD-tree is extremely simple since we can essentially just perform a ray/plane intersection test at each node.

Although the splitting strategy used above carves the space up in a uniform way like an oct-tree, this need not be the case for the kD-tree. At any given node we must always choose the correct axis aligned plane to split with, but that plane need not cut the volume into two equally sized child volumes. This is where a kD-tree generalizes the oct-tree, by allowing the world to be carved up into arbitrarily sized AABBs.

Figure 14.37 shows the same tree but with varying split plane positions being used for each child of a node. We can see that the root node (the red plane) splits its volume into two equal volumes as before. Both of its children split their volumes (the blue planes) at arbitrary positions along the depth of their volumes. We can also see that all the children at the 3rd level of the tree (the orange planes) have split planes at different heights from one another.

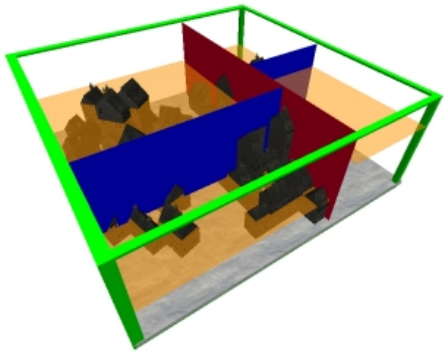


Figure 14.37

A useful characteristic of the kD-tree is that it allows us to favor subdivision in areas of the level that are more densely populated with geometry while still maintaining a balanced tree.

We now have a basic understanding of what a quad-tree, oct-tree, and kD-tree is, even if we are not yet intimate with the concepts behind coding each of these tree types. We are almost ready to start studying the implementation of our tree system, but before we do we will discuss the importance of tree balance, how to clip and split polygons to nodes, and explore exactly what T-junctions are and how we can repair them. Once done, we will be ready to start examining the code to all our tree types.

14.6 Mesh Trees and Polygon Trees

Whether you decide to store raw polygon data at the leaves of a spatial tree or just store whole meshes (or even clusters of meshes) is a decision you will have to make based on what the needs of your application are. Let us first have a quick discussion of the options.

14.6.1 Polygon Trees (Clipped/Unclipped)

If the geometry you intend to store in your tree is a totally static scene (e.g., a large static indoor level), then subdividing the scene at the polygon level might prove advantageous. This is especially true when using the tree in the broad phase of a collision system. A single leaf node will more accurately describe only the polygons that are contained within it, providing the ability to reject trivial data during polygon queries. A leaf node will contain only the polygons that are stored within the bounds of that leaf and the only polygons you are interested in intersection testing if the swept sphere is contained inside that leaf during a collision test. For rendering purposes, when a leaf is inside the frustum, the data we render associated with that leaf contains very few potentially non-visible polygons, even if the polygon data has not been clipped to the leaf nodes.

Obviously, if the polygon data has been clipped to the planes of the leaf node, then a leaf node will always contain an exact set of polygon data that fits inside that leaf node. Building the tree at the polygon level is obviously much slower than at the mesh level because every individual polygon has to be sent down the tree until it pops out in a leaf node and is added to its polygon buffer. If clipping is

being used then this process is slower again, and more polygons will be produced as a result. Certainly, the per-polygon approach is not ideal if the scene is largely dynamic because whenever geometry is moved, the tree will need to be rebuilt from scratch. This could potentially take an unacceptably long time and will not be practical to do in real-time between frame updates. If the scene is static however, then this is not a concern and we can benefit from the more exact fitting of the polygon dataset.

14.6.2 Mesh Trees

When building trees at the mesh level, tree construction is much quicker. We are no longer concerned with the individual polygons that a mesh is constructed from, but instead, we just use its world space bounding volume to determine which leaf nodes the mesh belongs in. This is extremely quick to do because even if a mesh contained 10,000 polygons, when traversing the tree to determine which leaf nodes the mesh belongs in, we are simply doing an AABB/AABB test at each node until we locate all the leaves the AABB intersects. The bounding volume test we perform when classifying the mesh against the child nodes depends on the bounding volume we are using to represent our meshes. If we are representing the bounds of our meshes with a bounding sphere for example, then determining whether a mesh intersects the bounding volume of a child node becomes a simple Sphere/AABB test. If the mesh's bounding volume is represented as an AABB (as ours will be), then the mesh/node test is an AABB/AABB test. As you might image, this is quite a bit faster than performing a Polygon/AABB test for each node and for every polygon in the mesh.

As an example, we might have a level constructed from 100 meshes, each containing 1000 polygons. If we were to compile our oct-tree or quad-tree at the polygon level, compiling the tree would mean traversing the tree with 100,000 polygons until they all eventually ended up being stored in their respective leaf nodes (perhaps with lots of clipping being performed). This is the approach we used in the quad-tree example earlier and as you can imagine, an awful lot of Polygon/AABB tests have to be performed at each node before the tree is fully compiled. If we decided instead to simply build the tree as a mesh tree, then all we would need to do to build the tree is send in the 100 bounding volumes of our meshes. It is the bounding volumes that would be classified against the nodes of the tree and would eventually end up in leaf nodes. In a tree used purely for the partitioning of mesh data, the leaf structure could contain a linked list or array of all the meshes stored in it.

> **Note:** It is common for a mesh that is large or just situated very close to the split planes to be assigned to multiple leaf nodes. There are strategies that exist to minimize this problem and guarantee that a mesh fits completely in a single node. Thatcher Ulrich's discussion of 'loose' oct-trees in *Game Programming Gems 1* (2001) would be a worthwhile read if you are interested in exploring this further.

A mesh tree is ideal for entities that are constantly having their position updated in the scene. If a mesh moves in our scene, we can simply unhook it from the leaf nodes to which it is currently assigned and feed it in at the top of the tree, and traverse the tree again with its bounding volume until we find the new leaf nodes that its bounding volume intersects. A mesh oct-tree for example might be ideal for a space combat game for example, where space could be uniformly subdivided into cubes (leaf nodes) and as a space ship mesh moves around the game world, the tree is traversed again to find the new leaf nodes it is contained within. A mesh is only rendered if one of the leaves it is contained in exists inside the frustum; otherwise it (along with all its polygons) is rejected from the rendering pipe.

When performing intersection queries on the tree (e.g., testing our swept sphere against the tree) the polygons of a mesh only have to be individually tested for intersection if the intersection volume intersects the bounding volume one of the leaf nodes in which the mesh is assigned. Since an entire mesh might exist in many leaf nodes simultaneously, it is important that the system establish some logic to avoid rendering the mesh more than once or checking its polygons for intersection more than once during a single query. As mentioned, this can also be a problem with a polygon tree when the polygons have not been clipped to the nodes. However, this is a relatively smaller problem with the polygon based tree since it would typically be only a handful of polygons here and there that would be rendered or queried multiple times. This might be something we are prepared to accept since it would probably not impact performance by a significant amount in the typical case. For a mesh tree however, it is crucial that this problem be resolved. The bounding volume for an entire mesh may well span dozens of leaf nodes, and if all those leaf nodes were visible, we certainly would not want to render the entire mesh with its thousands of polygons multiple times. The same is true for polygon queries. If our swept sphere intersected dozens of leaf nodes which all contained the same 20,000 polygon mesh, we certainly would not want to perform the swept sphere/polygon intersection tests for 20,000 polygons more than once, let alone dozens of times. Suffice to say our game would become less than interactive at that point.

14.6.3 Mesh Trees vs. Polygon Trees

It is easy to be seduced by the ease with which a mesh tree can be constructed and the speed at which the tree can be dynamically updated when objects in the scene move. To be sure, in many cases it will absolutely be the right choice for the job, whether you are creating a quad-tree, oct-tree or kD-tree (or any other type of tree – like a sphere tree, which is also very popular for dynamic objects). Of course, this decision will also be based on the format of your input data. If the scene is built from a set of mesh objects then the mesh tree would be much easier to implement. If the scene is represented as a static world space polygon soup (a little like the static geometry imported from an IWF file) a polygon tree is probably the best bet.

We do have to be aware of the disadvantages of a mesh tree however, since it may not always be a better choice than a polygon tree if the scene is comprised of multiple static meshes. Ease of construction and update speed does not always come without a cost. Whether that cost is significant depends of the specifics of your application. When you have dynamic objects in your scene, they absolutely have to be connected to the tree at the mesh level, so that they can have their positions within the tree updated in an efficient manner. When you have a scene represented as a static polygon soup, that will most likely fit in well with the polygon level subdivision techniques we have discussed. However, when the scene is comprised of multiple *static* meshes which do not need to have their positions updated, should a mesh tree or a polygon tree strategy be used?

To understand the disadvantages of a mesh tree, let us imagine a situation where a single large mesh consisting of 20,000 polygons spans dozens of leaf nodes (perhaps a terrain mesh). To be sure, this is a worst case example, but it will help to highlight the potential disadvantages of the mesh tree at performing frustum culling and polygon queries. Now, let us also imagine that we have a ray that spans only two leaf nodes which we wish to use to query the tree. In other words, when the ray is sent down the tree to retrieve the closest intersecting polygon, ideally only the polygons that exist in those two leaf

nodes would need to be tested. With the mesh tree, the leaf nodes might contain indices or pointers to the mesh objects that are attached to that leaf. When a leaf is found to be relevant to a query, every polygon in every mesh attached to that leaf will need to be queried at the polygon level. In the case of our large example mesh, only a very small subset of polygons (say 100) actually reside within those two leaf nodes, but we have no way of knowing that with a mesh tree. We simply know that some of the mesh's polygons may intersect the ray. Therefore, we would have to perform per-polygon intersection tests on the entire mesh (all 20,000 polygons) even though only a handful are within the space represented by the leaf nodes in which our ray currently resides. That certainly is not good for performance, and if this mesh was not dynamic, it would probably be wise for a broad phase collision implementation to partition this mesh at the per-polygon level.

> **Note:** The mesh itself could also be internally managed using some form of partitioning structure, even though it exists in a higher level scene mesh tree as single object.

In the case of a polygon tree where we have not performed clipping of the polygons, things are certainly a whole lot better as only a very small subset of polygons of the original mesh will be assigned to those two leaf nodes. This might not be an exact set of polygon data that fits inside the leaf nodes because some polygons may only be partially inside, but as long as we test those polygons and make sure that we do not test a single polygon multiple times if it exists in multiple leaf nodes, we only have to query the polygons that are (to some extent) inside the bounding volumes represented by the leaf nodes.

In the case of a polygon tree where clipping has been performed, we only ever query or render polygons that are completely contained within the leaf nodes that are intersected by the ray. The following diagram illustrates the disadvantages of using a mesh tree in such a situation where a ray/polygon query is being performed on the tree and the mesh is large enough to span multiple leaf nodes.

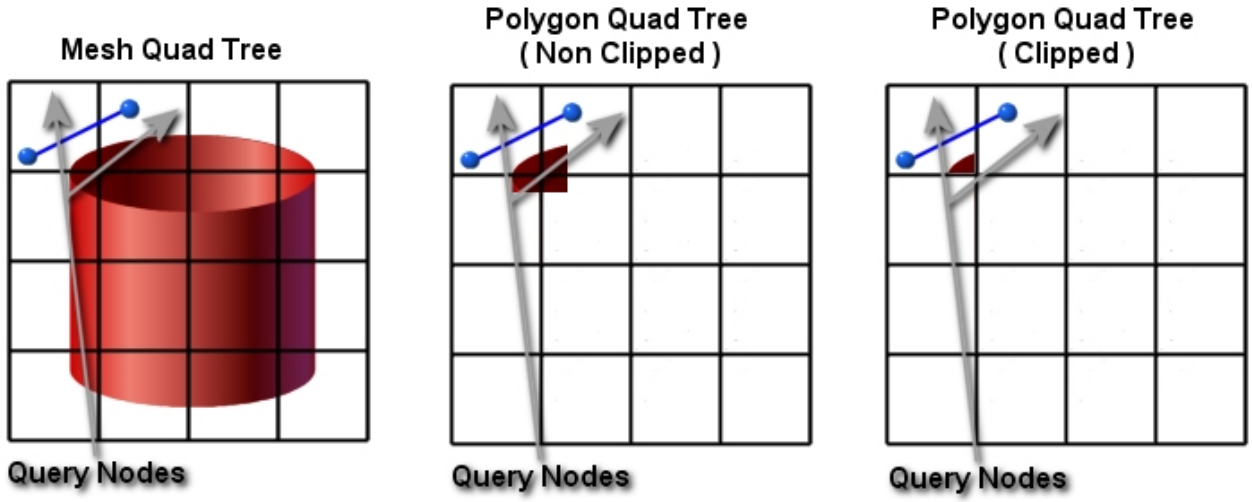


Figure 14.38

In this example we show a quad-tree as it is easier to represent on a 2D sheet of paper, but the same logic holds true for all trees. On the left we can see a single mesh of a cylinder consisting of our 20,000 polygons. As you can see, it spans a large region of the overall scene and therefore, is assigned to multiple leaf nodes. In this particular example the mesh is partially inside every single leaf node so would be assigned to each. The two blue connected spheres represent the ray being used to query the

scene and we can see in all three examples that it only ever intersects two leaf nodes in the top left corner of the scene. Ideally, only the polygons of the mesh that exist in those two leaf nodes should be queried. In the case of the mesh tree, we would have to test all the polygons of the cylinder for intersection, even though we can see that only a very small number of the mesh's polygons exist inside the two queried leaf nodes. If we were to render this mesh tree, the entire mesh would be transformed and rendered by the pipeline every single frame because the cylinder exists in every leaf node. This obviously means that this mesh would never be frustum culled and would always be rendered in full.

The middle image shows a non-clipped polygon quad-tree. Because the leaf nodes contain only the polygon data that are either inside or partially inside those nodes, only a very small portion of the mesh would need to be queried for collision detection with our ray. As you can see in this example, although the polygon data does not always exactly fit the leaf nodes (notice the overspill in the diagram) this does not affect the efficiency of our collision detection routines because only polygons that have some of their area inside the leaf nodes are tested. This would obviously be a lot quicker than querying every polygon in the entire mesh as was the case with the mesh tree. We can also see that during the frustum culling pass through the tree, most of the mesh's polygons will be rejected and only a very small subset would be rendered if the camera was positioned such that only a handful of leaf nodes existed inside the frustum. Some of those polygons may lie partially outside the frustum and would need to be clipped by the pipeline, but this would be negligible to performance in the typical case.

In the rightmost image we see the clipped version of the quad-tree where the leaf nodes contain only polygons that fit exactly inside them. If any polygon was found to be spanning multiple leaf nodes during tree compilation, the polygon would be clipped to those nodes and the fragments assigned to their relevant leaf nodes. This provides very little benefit during the polygon query phase as the same number of polygons would still need to be queried. The fact that those polygons have been clipped and are smaller has no affect on the speed of the ray/poly intersection routines. That is assuming of course that in the case of the non-clipped tree, provisions are made so that a polygon that spans multiple leaf nodes is not queried multiple times. We will use an application timer to assure that this is the case, which we will explain later in this lesson.

It is true when looking at the clipped polygon tree that less of the scene would need to be rendered because only polygon data that exactly fits inside the leaf nodes would be rendered if that leaf node is visible. The same number of polygons would still need to be rendered in the typical case and more polygons in the worst case where everything is visible (because of the splitting of polygons). We can imagine that if a triangle spanned a leaf node during the tree building process, that polygon could be clipped into two fragments, a triangle and a quad. We have now created three triangles where one previously existed. Frankly, on today's hardware, we would probably see little to no benefit from this type of tree versus the non-clipped case. In some cases, we may even see a *decrease* in performance due to the larger polygon count and increased number of DrawPrimitive calls. A clipped tree still provides benefits in other areas where we absolutely must know which section of a polygon lay inside a bounding volume, and you will see later that this is definitely the case when we create a node based polygon aligned BSP tree. For example, a non-clipped tree means that a single polygon may be assigned to multiple leaves. Thus, during a rendering pass we may render it multiple times. Sure, we can embed some logic that prevents a polygon that has already been rendered from being rendered again during a single traversal, but then this would introduce a per-polygon test. We want to render polygons in huge batches as quickly as possible and certainly do not want to be doing a per-polygon test to determine if

we should render each one. On modern hardware, it would generally be quicker just to render those polygons again. However, what about collision queries?

If we are using a non-clipped polygon tree for our broad phase collision system then we still have this problem to overcome. We certainly do not want to test the same polygon in the expensive swept sphere/polygon test more than once. This involves the costly transformation of the polygon into eSpace and the various intersection routines to determine if an intersection occurs. As discussed, we will implement a system so that the collision system can query non-clipped polygon trees without querying a single polygon multiple times.

So we have seen that while the mesh tree has some advantages, it certainly has disadvantages over its polygon based counterpart in other areas. The examples given above are somewhat extreme because if your meshes are of a size such that they fit inside a leaf node in their entirety, this situation will not arise. This situation only becomes problematic on a dramatic scale when the meshes in your scene can span many leaf nodes and have high polygon counts.

Object/Object Collision Testing

Spatial hierarchies can also speed up the collision detection between multiple dynamic objects in our scene. Let us imagine that we have created a space combat game and the region of space that our meshes will occupy has been spatially divided into uniform sized leaf nodes using an oct-tree. Our scene might very well have many spaceships all flying around in space, and while the polygon queries we have examined in this chapter show how a spatial hierarchy can be used to determine which polygons are intersected by rays or bounding volumes or swept spheres, what about the fact that every dynamic object in our scene might collide with every other dynamic object in our scene? In other words, collision detection does not have be performed only between our player and the static scene and any dynamic objects it may contain, but collisions can also be performed to make sure that the dynamic objects in our scene to not collide with one another.

It is quite common for dynamic objects to use simplified bounding volume collision detection in a game. In the interests of speed, when two objects collide in our world, we are very rarely interested in which polygons from each mesh intersected one another. Usually we will perform collision tests between meshes using bounding volumes such as spheres, OBBs or AABBs. Testing collision detection between two bounding volumes is generally much cheaper than testing two objects at the polygon level, which makes it ideal for a broad phase collision step for dynamic objects. If the bounding volumes of two objects do intersect, then you have two choices: you can either treat this as a collision and allow the meshes to respond to the collision accordingly, or you can use the intersection result as a test to see whether the meshes should be queried at the polygon level to make sure that an actual polygon/polygon intersection occurred between the two objects. You will usually find that using the bounding volume intersection result as a test for inter-object collision will suffice. Even if the objects did not physically collide (i.e., their polygons did not physically touch), everything is usually happening so quickly in a 3D game that the player will rarely notice the difference. Of course, the success of this approach depends on the bounding volume being used and how tightly it fits the actual object.

The quickest bounding volumes to test for intersection are spheres, but these are usually the bounding volumes that least accurately fit the shape of the object. When using a bounding sphere around a non-spherical object or an object that is much longer along one of its dimensions, the player may see two meshes respond to a collision of their spheres, even though it was quite visible to the player that the objects did not actually collide. If you are simply using the result of the bounding volume test to progress to the more accurate per-polygon testing process, then this is not so much of an issue. In this case, the sphere tests act more like a broad phase for objects that cannot possibly collide and therefore do not need to be tested at the polygon level.

A generally better fitting volume with fast intersection testing is the AABB. Therefore, one approach is to use an AABB for each of your objects and calculate it each time the player's position/orientation changes. While this might sound extremely slow this does not involve having to compile the AABB from scratch by testing each of the mesh's vertices whenever it rotates. This step only has to be performed when the AABB is first constructed and from that point on we can use some very quick and efficient math to recalculate a new AABB for the object in its new orientation. We will discuss such a technique later in the lesson but for now, let us assume that we are using AABBs for our dynamic objects for the remainder of this discussion.

One of the best fit bounding volumes is the Oriented Bounding Box (OBB). It is similar to an AABB in that it is also a box, but it differs in that it is calculated to try to fit the general shape of the object. Rather than be restricted to using the primary world axes, OBBs use a system closer to object local space to calculate the size and orientation of the box based on the general spatial distribution of vertices in the model. OBBs are a very popular choice for collision systems, especially ones that only wish to respond to collisions between bounding volumes and not perform actual polygon collision detection (although it can obviously be done if desired). Testing collisions between two OBBs is a fair bit slower than intersection testing two spheres or AABBs, but the results are going to be more accurate since the bounding volume is a better fit. They can certainly be worth it when the situation calls for it. Indeed an OBB (or multiple OBBs arranged hierarchically) can be constructed to bound a mesh so tightly that it allows us to do away with Polygon/Polygon' testing altogether in many cases and therefore greatly speeds up our ability to do more realistic collisions. Of course, there will still be times when the objects will react to a collision that did not actually happen between their physical geometry, even though their bounding boxes collided. However, this will likely go unnoticed by the player if the OBBs fit nicely. OBB construction and OBB intersection testing is fairly complicated from a mathematical perspective and takes more cycles to perform than simple AABB/AABB intersections. For the moment, we will hold off on discussing OBBs and revisit them a little later in the training series.

Now, let us imagine that we have decided that our space game is going to use a mesh tree and will use AABB/AABB intersection testing for the collision detection of its dynamic objects. Let us also imagine that in our space combat game a mighty battle is taking place between 500 ships. One approach to dealing with their collision status might say that for every frame, we loop through all 500 meshes and perform an AABB/AABB intersection test with the other 499 meshes in the scene. This seems to be a rather non-optimal approach since typically only a handful at most will actually be colliding. In fact, it is likely that most of these meshes will not even be in the same leaf nodes as any other ships and could not possibly collide at all.

This is where the hierarchy system can begin to show its true colors. The leaf structures in a mesh tree could contain a list of meshes that are inside it (or partially inside it) and depending on how much we subdivide our scene and the size of our final leaf nodes, there is a good chance that at most, a single leaf will contain only a few meshes. There is also a good chance that many leaves may contain only one mesh. Therefore, instead of naively testing every dynamic object against every other dynamic object, we could instead just test it against the mesh lists for the leaves the source mesh is contained within. For example, if the dynamic object structure could retrieve a list of leaves it is currently contained within, then we know the object could not possibly collide with anything that is not in one of those leaves. Therefore, we just have to access each of those leaves and test against the meshes stored in the mesh lists of those leaves.

Although we will be implementing our mesh tree strategy in a different way, the following code snippet demonstrates this concept. pDynamicObject is assumed to be an object that we wish to test for collision against other dynamic objects that exist in the same leaves. It is assumed to contain an array (LeafArray) of all leaf indices it is currently assigned to. Obviously, if any of these leaves contain only one mesh then this must be the only mesh in that leaf so no collision can occur there and we move on to process the next leaf it is contained within. Otherwise, we loop through and test the mesh's volume against every other mesh in the leaf using the ProcessCollision function. We might imagine how such a function would compare the AABBs of both meshes and perform some response if both objects collide. This is a very simple example using pseudo structures and functions and exists to give you a basic understanding of the way the information stored in a spatial hierarchy (especially a spatial tree that has the ability to also link dynamic objects to leaves) can be used to accelerate a multitude of processes.

```
for ( int i = 0 ; i < pDynamicObject->LeafCount; i++ )
{
  if ( pDynamicObject->LeafArray[i].DynamicObjectCount > 1 )
  {
        pLeaf = &DynamicOject->LeafArray[i];

        for ( int i = 0; i < pLeaf->DynamicObjectCount; i++ )
        {
            CDynamicObject *pTestObject = pLeaf->DynamicObjects[i];

            if (pTestObject!=pDynamicObject) ProcessCollision( pDynamicObject,
                                                               pTestObject );

        } // End for each object in current leaf

   } // If more than one object in leaf

} End for each leaf the dynamic object is currently in
```

Using the system described above, even if you had 10,000 meshes in your scene, if none of the meshes shared a single leaf node, no bounding volume collision tests need to be done. That is obviously much quicker than just blindly testing every object with every other object for collision. How much quicker? Well in the 10,000 mesh case you would have to do 49,995,000 tests! This is because you are calculating an addition series that amounts to the following formula:

$$\#tests = \frac{n(n-1)}{2}$$

If there are *n* objects, you do not test an object against itself and you only test between objects once (i.e., after testing A against B, there is no need to test B against A since the result is the same), thus cutting the number of tests in half. Even still, with a modest 50 objects in your scene, that is 1,225 tests that will need to be run. Granted, you will also have to weigh the cost of object insertion into the hierarchy versus the linear approach to determine which is more efficient for your system. However, if you are running the objects through the tree anyway (e.g., for their static collision testing) then you would not have to worry about adding overhead for traversals since that cost has already been incurred. In that case, the hierarchy will win out since you will only test objects that share leaves.

14.6.4 Combining Polygon Trees and Mesh Trees

So with the differences between the various tree types distinguished, which are we going to use in our lab project? Are we going to use a mesh tree, a clipped polygon tree, or a non-clipped polygon tree? Actually, we are going to allow our application to support all three tree types. We will have a single tree (such as a quad-tree for example) that compiles any static polygon/mesh data such that it is stored in the leaf nodes at the polygon level (polygon tree), but will also keep track of which leaves contain our dynamic objects (mesh tree). The basic building strategy will be as follows…

First we will load the data in from an IWF file. Any static data that is loaded from the IWF file (GILES™ internal mesh geometry stored in world space) will be added polygon by polygon to the spatial tree we are currently using. Once we have added all the polygons to the tree, the tree object will have a big list of all the static polygons we wish to have compiled into a spatial tree. The application will then call the tree's Build function which will instruct the tree to partition the space described by its polygon list. At the end of the build process, the space containing the polygon data will have been subdivided into a number of leaf nodes and each leaf will contain an array of the polygon pointers contained inside it. Further, we will allow the application to set a flag that instructs whether this polygon tree should be compiled using clipping. If not, then during the build process, a polygon will always be assigned to any leaf nodes it spans. If clipping is enabled, a polygon will be split if it spans leaf boundaries so that each leaf will only contain pointers to polygons that exactly fit within their volume.

It sounds like we have basically just decided to build a polygon tree, but that is not the full story. The leaf nodes of the tree itself will also be allowed to store pointers to dynamic objects (in actuality, a more generic structure will be used so that literally anything can be stored, but the end result will be the same). We can simply send the bounding volume of the dynamic object down the tree and store its pointer in any leaf nodes it is found to be contained within. We can also add some functionality to retrieve a list of these objects from the leaves as needed for various purposes (collision detection, rendering, etc.).

So we can see that our tree is basically both a mesh tree and a polygon tree although the tree will only directly store the static polygon data (meshes and other objects will simply be stored as pointers). In our

next chapter we will learn that the tree will even know how to render the polygon data. That is, in our main scene render function we can just call the ISpatialTree::DrawSubset function and it will only render the polygons contained in the leaves which are visible and have polygon data belonging to that subset. Since the objects can easily find out which leaves they are contained within, determining their visibility prior to rendering will also be quite straightforward.

Although this might all sound a little complicated you will see when we examine the code that it really is not that bad. In prior applications we had a single rendering pool, an array of CObjects which had to be rendered. Now, we essentially have two pools. We have a tree to render (which will contain all the static polygon data) and the array of CObjects (although they are ultimately linked into the tree as well, in a manner of speaking). However, only dynamic objects will be stored in the scene's CObject array from now on. All static geometry will be contained inside the tree and rendered/queried using the tree's rendering/querying methods. The tree will know how to render its polygon data efficiently and in a hardware friendly manner.

The tree will also expose methods such as ISpatialTree::CollectLeavesAABB. This method when passed an axis aligned bounding box will return a list of leaves intersected by that AABB. Functions like this can be called to run queries on any number of external objects to query their position within the tree even if the tree is not aware of their existence. Dynamic objects and terrain blocks will all use this function to determine whether or not the regions of space they are contained within are currently visible without their geometry ever having to be compiled into the tree. We could build a pure mesh tree (a space scene for example) by compiling an empty tree (subdividing empty space into a fixed number leaves) and then associate dynamic objects and terrain blocks with their leaves using this function.

> **Note:** The lab project that ships with this chapter imports its data from an IWF. The internal mesh data is treated as static polygon data and is compiled directly into the tree. Any external X file references will (as always) be treated as dynamic objects, loaded into CActors and stored in the scene's CObject array as usual. However, they will still be tracked by the tree itself. If you have a static X file that you would like to have compiled into a tree at the polygon level, then simply import it into GILES™ and export it as an IWF file. The X file data will become static geometry in the IWF file. If you have dynamic X file objects that you would like to place in the scene then place them in GILES™ as reference entities. Our application loading code will then know not to compile them into the tree at the polygon level and they will be treated as dynamic objects.

14.7 Areas of Interest

Before we start to examine the code there is one more matter that we must discuss and make provisions for if we are to make our tree classes as useful as possible. We must allow for the spatial tree to be instructed to partition space where perhaps no geometry exists during the compilation process.

Up until now we have discussed that during the recursive building process we will decide to stop subdividing nodes down a given branch of the tree if that node's bounding volume is very small or if only a small number of polygons exist there. These settings are all going to be configurable, so you may decide that you wish to have leaves with the capacity to store 1000s of polygons or perhaps only a few dozen. The same is true with the minimum node size setting. You may decide that you would only like

to only modestly subdivide your scene and set the minimum leaf size (the size at which a node is no longer further partitioned) to be a rather large volume of space. Alternatively you may instead decide to make the tree partition space is to lots of very small leaves. The choice is yours and trial and error will often be as good as any deductive reasoning when deciding for a given level what you strategy should be. However, there is still a problem with this procedure which, if nothing was done to remedy it, would make our tree classes useless as mesh only trees.

The problem is that the bounding boxes of the nodes (including the root node) are computed during the build process by finding the tightest bounding box that fits around the polygon data living in that node. In the case of the root node, this means that the bounding box will be calculated using the static polygon data that will be stored in the tree itself. We do not bother factoring the positions of any dynamic objects at this point for a few reasons. First, building the tree based on the positions of objects that will change in the very next frame makes little sense. Second, it is quite common in a game level for dynamic objects to be spawned on the fly in response to some game event, which means the dynamic objects would not even be available during the building of the tree. Finally, we have also stated that with our design, the tree will have no knowledge of what polygons a dynamic object contains. We will provide a means for determining which leaves an object is in, but those objects will manage their own polygon information.

Of course, we do not always want our spatial tree to be as small as the static polygon data that will be fed in. This is definitely the case if we wanted to build an empty tree (i.e., partition empty space) and then use it to manage the collision querying and rendering of dynamic objects. The root node would have been passed no static polygon data, its bounding box would have zero volume, and it would be the only node in the tree because it has no polygon data and is infinitesimally small.

Consider once again the creation of a space combat simulation where ships will be flying around and attacking each other. Imagine that there are no planetary bodies, so all we have is an area of space that will, at some point, contain many dynamic objects. The obvious thing to do here would be to first partition the empty region of space into an oct-tree, for example. Essentially, we wish to create a tree that will subdivide the space that the ships will be permitted to fly around in, but at build time we wish this tree to be empty. That is, none of the leaf nodes will contain any static polygon data. Why? Because in this example there is no static polygon data; we simply want to partition space for the benefit of our dynamic objects. Once the tree was built, each time a dynamic objects position is updated, we can feed its AABB into the tree (using the ISpatialTree::CollectLeavesAABB method). This method determines which leaf nodes contain the dynamic object and store this information internally (as well as export it to the caller).

Figure 14.39 shows how three dynamic objects might look when placed within the region of space subdivided by the oct-tree. Remember, the oct-tree itself would still think it was an empty tree as it has no polygon data stored at its leaf nodes, but it does update itself each frame and determines which leaves are currently inside the frustum and are therefore visible. Once the dynamic objects are introduced, it can keep track of this information internally using some generic data structures and return the list of visible leaves as needed, or even return the list of visible dynamic objects if desired. Incidentally, as Figure 14.39 also shows, there would be no need to test the dynamic objects against each other for collision as none of them exist in the same leaves.

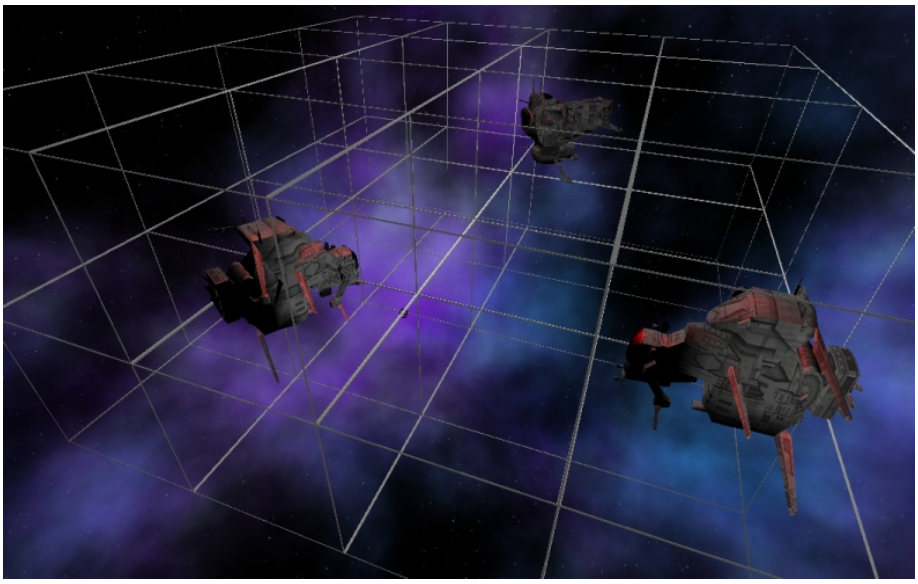


Figure 14.39

So we can see that we need the ability to create empty trees of a specific spatial size specifically so that we can control the region of space that will become part of the tree, even if that region is not described (or fully described) by the static polygon data from which the tree is generally built. Even if there is static data with which to build the tree, we might not want to simply compile only this exact region into the tree. We still may want the sky above a cityscape model to be partitioned and stored in the tree so that flying objects can enter the cityscape's airspace and benefit from the tree as well.

In the example shown in Figure 14.39 we might pass a large AABB into the tree building function which specifies a region of space that must become part of the tree and be spatially subdivided. We refer to these in our system as *detail areas* (or *areas of interest*). The tree can then continue to carve this space just as if there was polygon data contained in it. Only in this instance, the leaf nodes would contain no polygon data.

Another example where this is useful is when using a scene that contains a CTerrain object. As we know, a CTerrain object is constructed from a series of CTerrainBlocks where each block is a separate mesh. As terrain blocks could be quite efficiently frustum culled by testing their bounding boxes against the frustum it would be wasteful to store copies of the every triangle in the terrain in a tree. We saw in the previous lesson how our collision system essentially avoided doing the same thing by converting the swept sphere's AABB into terrain space and building only the relevant triangle data on the fly that needed to be tested. The same should also be true for our rendering; it would be overkill to store the triangle data for each terrain block in the tree when we could instead treat each terrain block as a dynamic/detail object. That is, we could pass the AABB of the of the terrain blocks down the tree and have pointers to them stored in the leaf nodes in which they belong. Only when those leaf nodes are visible does the terrain block render itself.

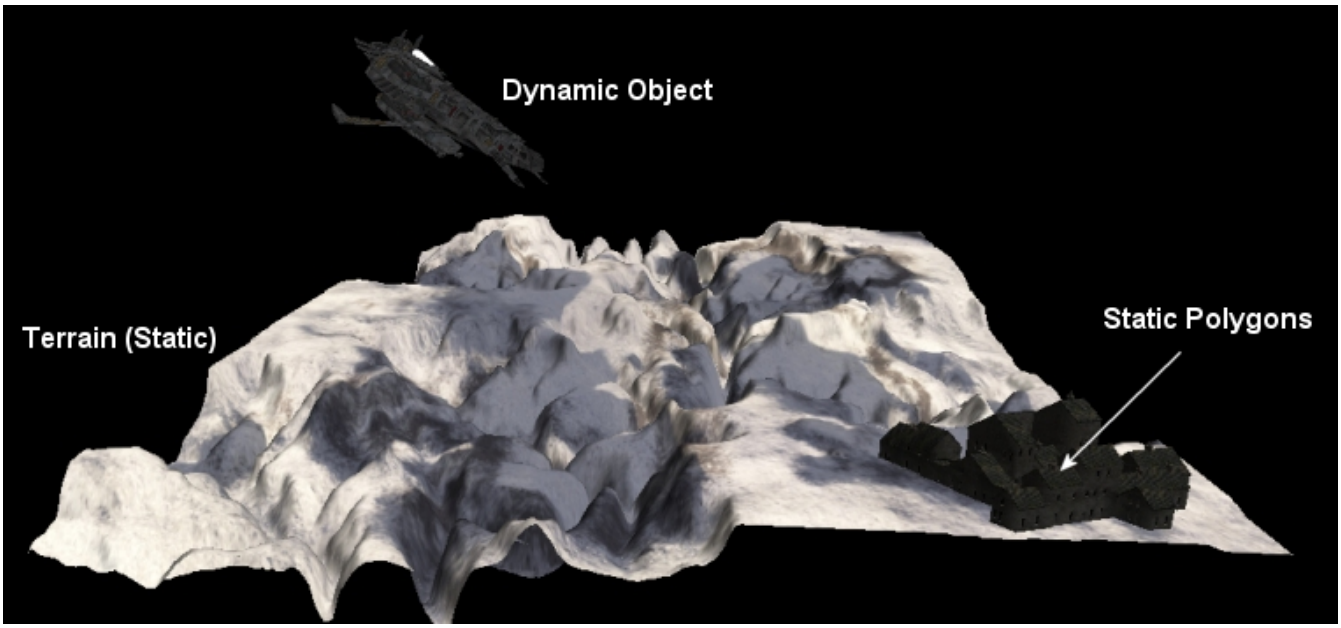


Figure 14.40

This is a good example of why we need the ability to add areas of interest (detail areas) to the tree. In Figure 14.40 we see a scene that contains a terrain (composed of multiple terrain blocks), a dynamic object in the sky above the terrain, and a small settlement made using static geometry. Because we do not wish to assign the terrain triangle data to the tree and we do not wish to factor in any temporary position of the detail object during the tree building process, the only polygons initially input to the tree building phase would be the static polygons labelled on the image. However, using the building strategy we have discussed thus far, the root node of the tree would have its bounding volume calculated only to be large enough to contain those static polygons. In Figure 14.41 we show that this would create an oct-tree that actually subdivides only a small section of the overall space in the scene we wish to use.

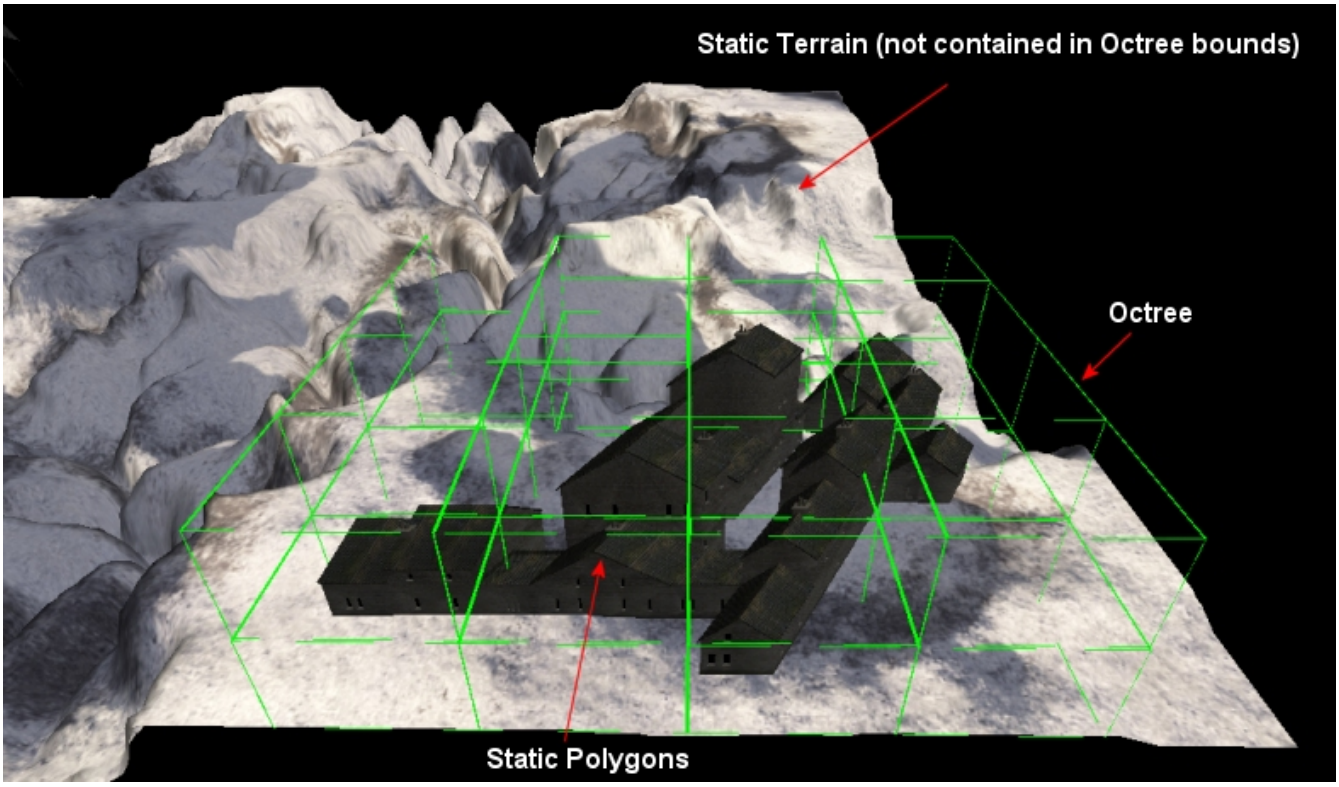


Figure 14.41

Now imagine that after the tree has been built, we decide to send the bounding volumes of each terrain block down the tree to find out in which leaf nodes they belong. We have a problem -- most of the terrain blocks are not even within the bounds of the root node's volume, so they would be found to be outside the tree and would not be able to benefit from the tree's various properties. The same is true once we place our spaceship (dynamic object) in the skies above the terrain. The detail object would have its AABB passed down the tree each time it is updated and we would find that it is contained in no leaf nodes since it is not within the area of space occupied by our partitioning scheme. This might mean the object does not get rendered at all (because it is technically not inside a visible leaf) or that such objects must be individually tested and rendered (depending on the rendering strategy you are using).

Alternatively, using the same detail area example seen above, we could pass into our tree building function a large bounding box that encompasses not only the stone settlement but also the terrain and all the sky above the terrain we intend to use for our dynamic objects. This would force the root node of the tree to be of a size that is large enough to contain the entire scene and each level in the tree would further subdivide this space.

If we imagine that we have done just that and that our tree building strategy is updated to calculate the correct size of a node's bounding box (not only on the static polygon data in its list, but also any detail area AABBs that have be registered with the tree object prior to the commencement of the build process), we can see the results would be exactly as needed as shown in Figure 14.42.

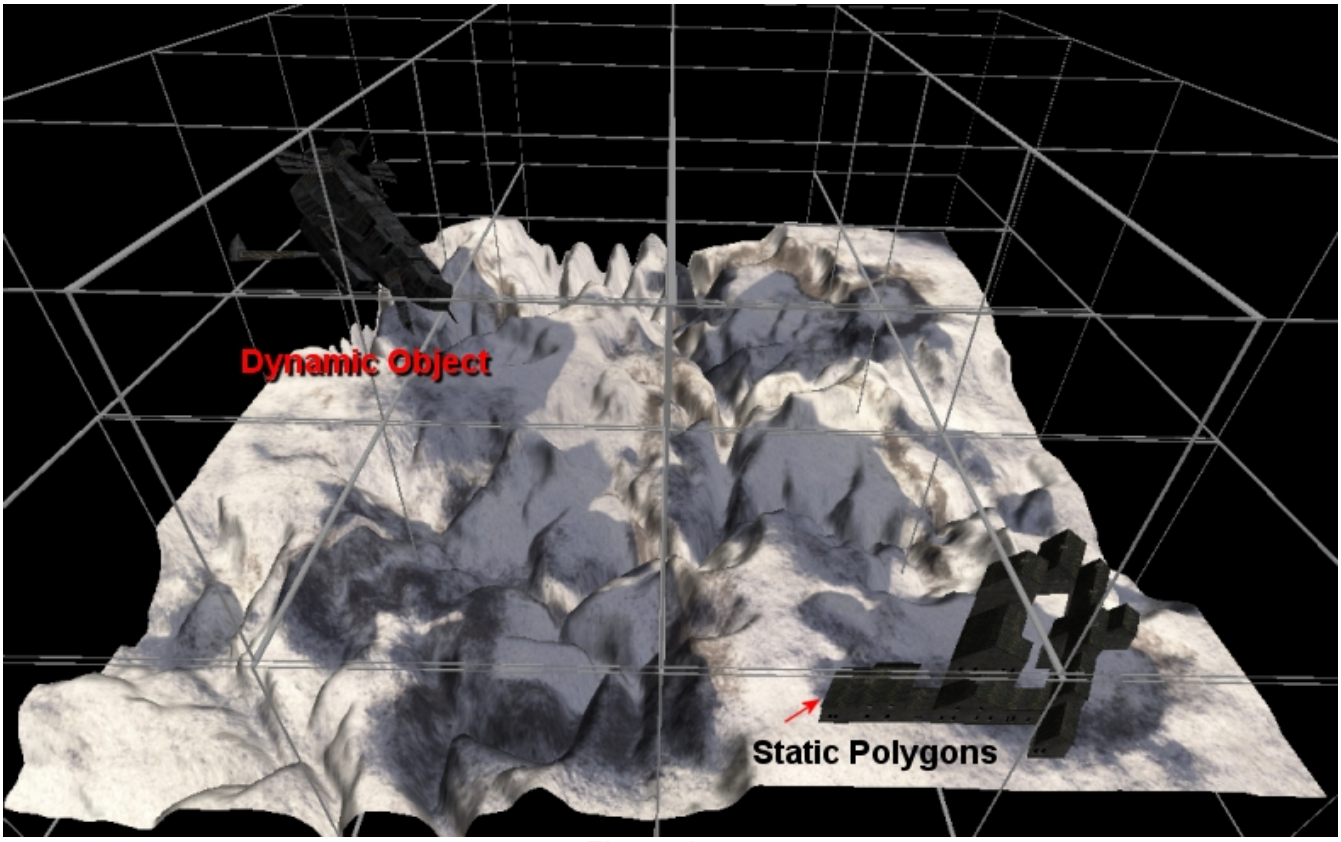


Figure 14.42

This really does solve all of our problems and allows us to build trees that either completely partition empty space or that partition space that is not necessarily fully occupied with static input polygon data.

Of course, this does not mean that we are restricted to passing only a single AABB as an area of interest. We might pass in multiple areas of interest at different positions in the level identifying areas where nodes should be partitioned even if no polygon data currently exists there. During the building process the condition used to determine whether a given node should be made a leaf node is now either of the following:

1. The bounding volume of the node is sufficiently small.
2. The bounding volume contains a small amount of polygon data *and no area of detail*.

The second condition (polygon count) has subtly changed, but it is an important one. Before we said that we would make a node a leaf if the number of polygons in that node's list is below a certain threshold amount. But now, even if there are no polygons in the node, but there is a detail area intersecting that node and the node has not been subdivided down to our minimum leaf size, we will continue to subdivide. This means that not only do these detail areas allow us to include space in the tree that is outside the region described by the static polygon set, but they also allow us to specify areas at any point in the level where we would want the space to be more finely partitioned down to the minimum leaf size.

It will be easy to add these areas of interest to our tree system. Our tree classes will expose a function called AddDetailArea which allows the application to register any bounding volumes with the tree as areas of interest. These areas must be registered prior to the Build function being called so that the construction of the tree can them in to its computations when determining the bounding boxes of each node and figuring out whether or not a node should be further subdivided. Further, we will also provide the ability to register a context pointer with the detail area which can point to application specific data. This could, for example, point to a terrain block that needs to be rendered or a sound that should be played whenever the player is in that leaf (or if that leaf is visible). Although we will actually have no need to use this context pointer in our first implementation, it is handy to have around and is something we will undoubtedly find useful moving forward.

14.8 Tree Balance

Before we begin to code, we must be aware of the implications of creating a wildly unbalanced tree. A perfectly balanced tree is a tree where all leaf nodes exist at exactly the same level in the hierarchy. Although it is rarely possible to achieve a perfectly balanced tree (while not subdividing empty space and introducing unnecessary nodes) it is something we wish to strive for as much as possible in many cases because it allows us to keep tree traversal times consistent during queries. The last thing we want is one part of our tree to be many levels deep and another section of the tree to be only a few levels deep. This will cause inconsistent frame rates when querying the deepest parts of the tree. Figure 14.43 shows an unbalanced quad-tree.

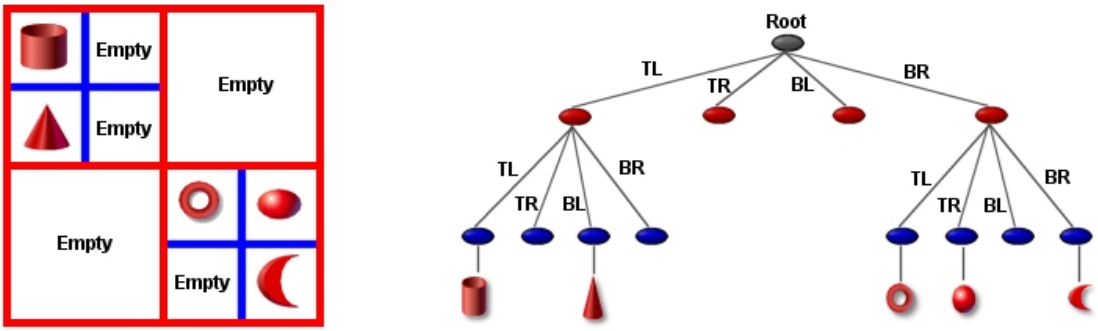


Figure 14.43 : Unbalanced Quad-tree

In this example we can see that only two of the four children of the root node have geometry to subdivide and as such, nodes TR and BL (which have no geometry passed to them) become empty leaf nodes (terminal nodes). However, the polygon data passed into the TL and BR children of the root do have geometry that falls within their bounds, so their volumes are further divided. Although this is a simple and small example, we can see that queries would take longer to perform in the top left and bottom right quadrants of the scene than in the top right and bottom left quadrants. That is, we would reach the terminal nodes quicker for the top right and bottom left quadrants. The difference in query times in the real world is much greater when we deal with trees that are many levels deep. It is better to have a scene that can be rendered/queried at a consistent 60 frames per second than it is to have a scene where the frame rate runs at 150 frames per second in some areas and at 10 frames per second in others.

This is a somewhat drastic example but the point is that the application will benefit from smoother physics, collision response, and general movement if the frame rate can stay relatively consistent throughout.

If we imagine that we had the ability in the quad-tree to slide the clip planes to arbitrary positions at each level, we might imagine that we could end up reducing this to a two level tree where all four children of the root (2^{nd} level) has mesh/polygon data of its own. This is essentially what can be achieved using a kD-tree -- choosing a split plane at any given node that is positioned such that it splits any remaining geometry into exactly two pieces, even if all that geometry is located on one side of the parent nodes volume.

> **Note:** One possible method for choosing a split plane is to divide the polygon data into two equal sets at a given node. If splitting along the X axis at a given node for example, we could sort all the vertex data by their X component and then build a plane that passes through the median vertex.

kD-trees are generally much deeper trees than oct-trees if you intend to get the same level of spatial partitioning because they only divide space into two spaces at each node, so we will usually have to traverse deeper into a kD-tree. However, each level in the tree has fewer child nodes assigned to it than its equivalent level in an oct-tree, so traversal is cheaper at each level. In the above example, we see a quad-tree that has four nodes on its second level and eight nodes on its third level. The kD-tree on the other hand would have only two nodes on its second level and four nodes on its third level. Therefore, the deeper tree traversal required in the kD-tree case is offset by the speed at which the kD-tree can traverse its tree (to some extent). For example, we may find when testing a bounding volume against the split plane stored in the root that the query volume is contained in the front halfspace of the clip plane. With one plane/volume test we have just rejected half of the entire scene. In an oct-tree, each AABB/AABB test we perform will reject only $1/8^{th}$ of the parent node's volume.

Choosing a split plane at a kD-tree node is not simply about finding the plane that best balances the tree since we sometimes have other considerations (perhaps more important ones) that we wish to factor in. For example, a common goal when choosing a split plane is to find one that produces the least number of polygons that will have to be split (if we are building a clipped tree) in order to fit into the child nodes. Every time we split a polygon into two pieces we increase the polygon count of our scene and this could grow significantly if split planes are being chosen arbitrarily. Furthermore, clipping polygons arbitrarily also often introduces a nasty visual artifact called a T-junction in the polygon data. We will discuss T-junctions later, but for now just know that due to the rounding errors accumulated in the rendering pipeline, two polygons that are neighbors (i.e., they look like one polygon), but that do not have their vertices in identical places along the shared edge, can produce a sparkling effect. Essentially, because their vertices are not in identical places, there are sub-pixel gaps produced during rendering. This results in odd pixels between the two polygons (along the seam) that do not get rendered and anything rendered behind it can show through. You have probably seen this artifact in some video games -- it is commonly referred to as *sparkling*. T-junctions also wreak havoc with vertex lighting schemes as we will find out later.

T-junctions often occur whenever a great deal of polygon clipping is employed on a scene, so these artifacts can manifest themselves with all the tree types we have discussed thus far (or indeed with any clipping process). However, when we are using uniform partitioning (such as with the quad-tree or oct-tree) we generally create many fewer T-junctions in our geometry (this will make sense later when we

cover T-junction repair). When using a kD-tree that is allowed to arbitrarily position its split plane, T-junctions artefacts are common. Although T-junctions can be repaired after the tree has been compiled (we will cover how to do this later in the lesson) the repair of a single T-junction introduces an additional triangle into the scene. Therefore, in the case of a kD-tree that is using non-uniform split planes, hundreds or maybe thousands of T-junctions could be produced by the tree building process. The repair of these T-junctions would introduce hundreds or thousands of additional triangles in our scene, which is far from ideal. The existence of T-junctions are only a concern if the polygon data contained in the tree is intended to be collected and rendered. If the tree polygon data is only being used for collision queries, we can typically leave the T-junctions alone since they will not affect our collision queries.

Because our tree class will also be used as a render tree in addition to a collision query tree, we will force our kD-tree to always split into two equally sized child volumes at any given node. This will hopefully reduce the number of T-junctions produced during the building of the tree and keep our polygon count from growing too large during the T-junction repair.

> **Note**: Do not worry too much about what T-junctions are for now as they will be discussed over the next few sections. They are only mentioned now as a justification for why our kD-tree implementation will always use a split plane at each node which partitions the node's volume into two uniformly sized children.

So the balance of a tree is important but as noted, it is not our only goal when compiling our trees. Often trial and error will produce the best results and this is where benchmarking your code is ***very important***. You may find for example that using an arbitrary split plane at a kD-tree node creates a more balanced tree at the cost of many triangles being inserted into the scene to repair the T-junctions produced. This increase in polygons could create a consistently deeper tree and undo much of what you were trying to achieve by balancing the tree in the first place.

We have now discussed the theory behind the oct-tree, the quad-tree, and the kD-tree and we are ready to start looking at the code to both the tree building processes as well as the support structures, routines, and intersection queries that will need to be implemented. We will delay our discussion of the BSP tree until Chapter 16, after we have stepped through the source code needed to build the three tree types we have already discussed. We will also discuss some of the core mathematical routines we will need to run queries on our tree. The BSP tree, although constructed in a similar way, ultimately exhibits very different characteristics due to the fact that it divides space arbitrarily. The application of a BSP tree is often done to achieve a different goal than just spatial subdivision as we will see later in the course.

In the remainder of this textbook, we will cover all the code that builds the quad-tree, oct-tree, and kD-tree. Ultimately we will integrate them into our collision system as a broad phase suite. Building a tree rendering system that is hardware friendly is rather complex and will be discussed in its own chapter. Lab Project 14.1 uses the rendering system that will be discussed in the following lesson, so you should probably ignore most of the tree rendering code for now.

14.9 Polygon Clipping

Several times throughout this chapter we have mentioned that our system will have the optional ability to create clipped trees. That is, trees that are constructed such that the polygon data stored in each leaf fits completely in its volume. This means that the tree building process will involve a good amount of clipping of the original polygon data to the split planes that subdivide the node into its children. This section will discuss the process of clipping polygons.

If you have never implemented code that clips a polygon to a plane, you will probably be pleased to learn how simple it actually is. The basic process is one of stepping around each edge of the polygon and classifying its vertices against the plane in order to build two vertex lists that will describe the front and back split polygons (i.e., the polygon fragments that lay on each side of the plane). If the current vertex being processed is in front of the plane and the following vertex is also in front of (or on) the plane, then the current vertex belongs in the front split polygon and is added to its vertex list. Likewise, if the current vertex being processed is behind the plane and the next vertex in the list is also either behind or on the plane, then the current vertex being processed is added to the vertex list of the back split polygon.

If however, the current vertex being processed and the next vertex in the list are on opposing sides of the plane, then we have found an edge in the original polygon where an intersection occurs with the plane. In this instance we use the two vertices in the edge to create a ray and perform a Ray/Plane intersection test with the split plane. The result of this intersection test will be the position on the plane at which the ray intersects it. The vertex in the edge that was in front of the plane is added to the front split polygon and the other vertex in the edge located on the back side of the plane is added to the back split polygon. The intersection point where the ray intersected the plane is made into a new vertex and is added to *both* the front split and the back split polygons.

After having done this for every vertex/edge in the original polygon, will have two new polygons and the original polygon can be discarded. Obviously, if you do not intend to the split the polygon into two fragments, but are instead interested only in clipping polygons to a plane, you can simply discard the vertices on the half space of the plane that you are clipping away and return only a single polygon. Thus, a polygon splitting function could easily double as a polygon clipping function by simply being instructed to discard any vertices located in a certain plane halfspace. We do a similar thing in our polygon clipping function. The function will accept as parameters the original polygon that is to be split/clipped and will also be passed pointers to two polygons that, on function return, will be filled with the front split and back split polygons. Passing NULL as one of these parameters will turn the splitter function into a clipper function. If you pass NULL as the parameter where you would normally pass a pointer to a polygon that will receive the back split data, the function will recognize that you have no interest in the vertex data that is behind the plane and simply clip the polygon to the plane and return only the section of the polygon that is in the plane's front halfspace.

Figure 14.44 shows how a pentagonal polygon would be split into two by an arbitrary split plane.

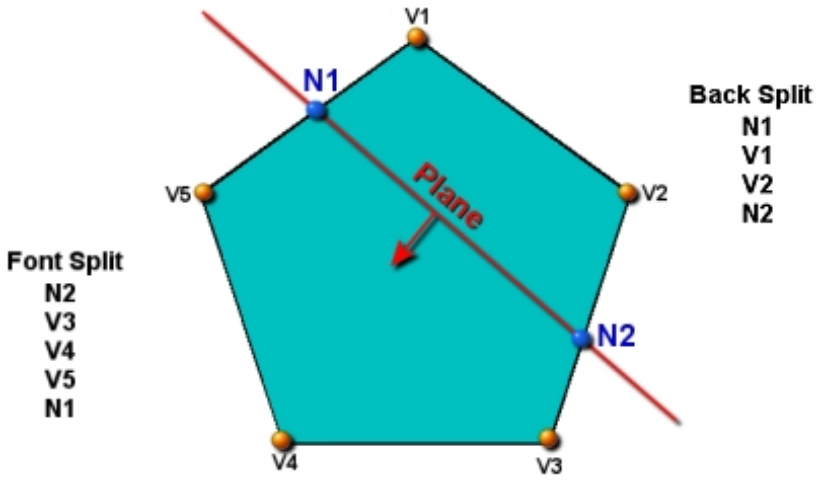


Figure 14.44

Let us step through this example using the technique described above. The original polygon has five vertices labelled v1 – v5 as shown. The plane can be seen as the red line in the diagram clearly carving the polygon into two parts. We would set up a loop starting at the first vertex v1. We would classify this point against the plane and discover it is in the back space of the plane. If v2 is also in the plane's back space, then a new polygon structure is allocated for the back split and v1 is added to its vertex list.

In the next loop iteration we check vertex v2 which is also in the plane's back space. If v3 is also in the plane's back space, then v2 would be added to the back split polygon also. However in this case we can see that v3 is actually in the plane's front space so we know for sure that the polygon edge (v2,v3) intersects the plane. Therefore, we create a ray from v2 to v3 and perform a ray/plane intersection test. This will return the position at which the ray intersects the plane, shown as N2 in the diagram. We add v2 and N2 to the vertex list of the back split polygon and add the vertex N2 to the vertex list of the front split polygon as well. The back split polygon now has vertices v1,v2, and N2 in its vertex list and the front split polygon so far has a single vertex, N2.

Next we test vertex v3 and find it is in the plane's front space. Because v4 is also in the front space, v3 is added to the vertex list of the front split polygon so that now it contains two vertices, N2 and v3. Next we test vertex v4 which is in the plane's front space also. v4 is added to the front split polygon's vertex list because the next vertex v5 is also in front of the plane. The front split polygon now has vertices n2, v3, and v4 in its vertex list.

Finally, in the last iteration of the loop we classify vertex v5 against the plane and discover it is in the plane's front space. However, vertex v5 is the last vertex in the original polygon and it forms an edge with the first vertex v1, so we must test that this edge does not span the plane. We discover that v1 is in the back space of the plane and therefore the edge (v5 ,v0) does indeed span the plane. Thus, we create a ray from v5 to v1 and perform a ray/plane intersection calculation to return to us the intersection point N1 where the ray intersects the plane. v5 and N1 are added to the front split polygon and N1 is also added to the back split polygon and we are done. In this example, the front split polygon would have vertices N2, v3, v4, v5, and N1. The back split polygon would have vertices v1, v2, N1, and N2. We now have two polygons with a clockwise winding order which can be returned from our polygon splitting function. At this point, the original polygon can be discarded.

14.10 Implementing Hierarchical Spatial Partitioning

In order to split and clip polygons, certain utility functions will be added to our CCollision class as static methods. We will add methods to help our clipping routines as well as new bounding volume intersection methods that will be used by the tree during traversals. The CCollision class has very much become our core library of intersection and plane classification routines.

When discussing the process of polygon splitting in the last section it became apparent that much of the process involves the classification of each vertex in the source polygon against the split plane. This is so we can determine whether a given vertex belongs to the front split or the back split fragment. Although we have discussed several times in past lessons how the classification of a point against a plane is a simple dot product and an addition in order to compute the plane equation, we have wrapped this operation in a method called PointClassifyPlane in our CCollision object. Instead of just returning the result of the dot product, the function returns one of four possible members of the CLASSIFYTYPE enumerator shown below (defined in the CCollision namespace):

Excerpt from CCollision.h

```
// Enumerators
enum CLASSIFYTYPE { CLASSIFY_ONPLANE = 0,
                    CLASSIFY_BEHIND = 1,
                    CLASSIFY_INFRONT = 2,
                    CLASSIFY_SPANNING = 3 };
```

As you can see, it is much nicer during our building routines to get back a result that clearly is behind, in front, on plane, or spanning the plane rather than us having to determine this ourselves using the sign of the dot product result. The PointClassifyPlane function will only ever return either CLASSIFY_ONPLANE, CLASSIFY_BEHIND or CLASSIFY_INFRONT as it is impossible for a single point in space to be spanning a plane. However, the CLASSIFY_SPANNING member will be returned by other classification functions that we will discuss shortly. Let us now look at this new function (which is really just a dot product wrapper).

14.10.1 PointClassifyPlane – CCollision (static)

The PointClassifyPlane method accepts three parameters. The first is the vector describing the point we would like to classify with respect to the plane, the second is the normal of the plane, and the third is the distance to the plane from the coordinate system origin. The function then evaluates the plane equation by performing the dot product between the point and the plane normal (two 3D vectors) and then adding the resulting distance. This is obviously equivalent to performing AX+BY+CZ+D where ABC is the plane normal, D is the plane distance, and XYZ is the point being classified. The result of the plane equation is stored in the local variable fDistance.

```
CCollision::CLASSIFYTYPE CCollision::PointClassifyPlane
                         ( const D3DXVECTOR3& Point,
                           const D3DXVECTOR3 &PlaneNormal,
```

```
                            float PlaneDistance )
{
    // Calculate distance from plane
    float fDistance = D3DXVec3Dot( &Point, &PlaneNormal ) + PlaneDistance;

    // Retrieve classification
    CLASSIFYTYPE Location = CLASSIFY_ONPLANE;
    if ( fDistance < -1e-3f ) Location = CLASSIFY_BEHIND;
    if ( fDistance >  1e-3f ) Location = CLASSIFY_INFRONT;

    // Return the classification
    return Location;
}
```

Be default we assume that the point is on the plane by setting the local Location variable to CLASSIFY_ONPLANE and then go on to test the sign of the result of the plane equation. If fDistance is less than zero (with tolerance) then the point is behind the plane. If fDistance is greater than zero then the point is in the plane's front halfspace. At the bottom of the function the local Location variable will contain either CLASSIFY_ONPLANE, CLASSIFY_BEHIND or CLASSIFY_INFRONT which is then returned to the caller.

The above function will be used by our polygon splitting routine, but the following routine will be used by the core tree building functions to determine if a given polygon needs to be split. It is this next routine that can be used to determine whether a given polygon is in front or behind a plane, lying on the plane, or spanning the plane. In the spanning case we know we have a polygon that has vertices on both sides of the plane, which tells us the polygon needs to be split.

14.10.2 PolyClassifyPlane – CCollision (static)

This is another very useful function that you will likely find yourself using many times as you progress in your 3D programming career. For this reason, we have not built it into the tree code, but have added it to our collision library where we can continue to use it in the future without any dependency on anything else. Remember, these static members always exist even when an instance of the CCollision object does not. Thus CCollision represents a handy collection of intersection routines that can be used by the application even if the application is not using the actual collision system.

This function has a fairly simple task in that it is really just performing the test described in the above function for each vertex in the passed polygon. That is, we classify each point in the polygon against the plane and keep a record of how many points were found behind the plane, in front of the plane, and on the plane. After testing each vertex we can then examine these results to get our final outcome. For example, if the number of vertices found in front of the plane is equal to the vertex count of the polygon, then it must mean the entire polygon is in front of the plane. Likewise, if the number of vertices found behind the plane is equal to the vertex count of the passed polygon then the entire polygon must lay behind the plane. If all points lay on the plane then the polygon is obviously on the plane and we return that result. Finally, we know that if none of the above conditions are true then it must mean some vertices were behind the plane and some were in front of it and therefore, we return CLASSIFY_SPANNING.

Although our spatial trees will all store their static polygon data at the leaves of the tree in CPolygon structures (something we will look at in a moment) we really do not want any generic and potentially reusable routine that we add to our collision namespace to be so reliant on external structures. For example, in a future lab project we might not store our polygon data in a CPolygon object, or we may change the format of its internal vertices, which would mean in either case that we would not be able to use this function. Therefore, we have kept this function as generic as possible by allowing the application to pass the vertex data of the polygon being classified as three parameters.

The first parameter is a void pointer which should point to an array of vertices which define the polygon. This is a void pointer so that it can be used to point to any arbitrary array of vertex components. For example, this could be a pointer into a locked system memory vertex buffer or the vertex array of a CPolygon structure. As this function is only interested in the positional data of each vertex and has no interest in other vertex components, we could also just pass an array of 3D position vectors. The second parameter is the number of vertices in this array so that the function knows how many vertices it has to test. The third parameter is a stride value describing the size of each vertex in the array (in bytes). Remember, this function has no idea how many vertex components you have in your structure, so it has no idea how large each one is and how many bytes a pointer must be advanced to point to the next vertex in the array. Passing a stride value solves this problem and allows the function to step from vertex to vertex in the array without caring about what other data follows the positional data. As the fourth and fifth parameters we pass the normal and the distance of the plane which we wish to classify the polygon against. Below we present the complete code listing to the function, which you should be able to understand with very little explanation.

```
CCollision::CLASSIFYTYPE CCollision::PolyClassifyPlane(
                                                 void *pVertices,
                                                 ULONG VertexCount,
                                                 ULONG Stride,
                                                 const D3DXVECTOR3& PlaneNormal,
                                                 float PlaneDistance )
{
    ULONG   Infront  = 0, Behind = 0, OnPlane=0, i;
    UCHAR   Location = 0;
    float   Result   = 0;
    UCHAR  *pBuffer  = (UCHAR*)pVertices;

    // Loop round each vector
    for ( i = 0; i < VertexCount; ++i )
    {
        // Calculate distance from plane
        float fDistance = D3DXVec3Dot((D3DXVECTOR3*)pBuffer,
                                      &PlaneNormal ) + PlaneDistance;
        pBuffer += Stride;

        // Retrieve classification
        Location = CLASSIFY_ONPLANE;
        if ( fDistance < -1e-3f ) Location = CLASSIFY_BEHIND;
        if ( fDistance >  1e-3f ) Location = CLASSIFY_INFRONT;

        // Check the position
        if (Location == CLASSIFY_INFRONT )
            Infront++;
        else if (Location == CLASSIFY_BEHIND )
```

```
            Behind++;
        else
        {
            OnPlane++;
            Infront++;
            Behind++;

        } // End if on plane

    } // Next Vertex

    // Return Result
    if ( OnPlane == VertexCount ) return CLASSIFY_ONPLANE;     // On Plane
    if ( Behind  == VertexCount ) return CLASSIFY_BEHIND;      // Behind
    if ( Infront == VertexCount ) return CLASSIFY_INFRONT;     // In Front
    return CLASSIFY_SPANNING; // Spanning
}
```

In the above code we first cast the void pointer to a byte pointer so that the stride value will describe exactly how much we need to increment this pointer to step to the next vertex. We then set up a loop to loop through each vertex of the polygon. For each vertex we compute the plane equation for that vertex and the plane and store the result in fDistance. We then increment the vertex pointer by the stride value so that it is pointing at the next vertex in the array. Next we test the value of fDistance and set the value of Location to either CLASSIFY_ONPLANE, CLASSIFY_INFRONT or CLASSIFY_BEHIND depending on the result. Then we test the value of the Location variable and increment the relative counter. For example, if the vertex is behind the plane, then we increment the Behind counter. If it is in front of the plane we increment the InFront counter. Otherwise it means the vertex is on the plane and we increment the OnPlane, InFront, and Behind counters. It is very important that in the on plane case we also increment the InFront and Behind counters because these counters are trying to record how many vertices would exist in a front split and back split polygon were we actually creating one. Obviously, if a point is on the plane, it could belong to either a front split or a back split polygon. Using this strategy allows us to perform simple tests at the bottom of the function to determine if the polygon is considered in front or behind the plane. Remember, a polygon may still be considered to be behind a plane even if some of its vertices are touching the plane.

At the bottom of the main for loop we can see that we have stored the values of how many vertices to be found in each condition in the OnPlane, Behind and InFront local variables. If the OnPlane value is equal to the vertex count for example, then we know the polygon is completely on the plane. If the vertex count is equal to the number vertices found to be in front or behind the plane then it must mean the polygon lay in front or behind the plane, respectively. Finally, if none of these cases are true we return CLASSIFY_SPANNING since there were obviously vertices found in both plane halfspaces.

14.10.3 CPolygon – Our Static Geometry Container

Earlier we noted that our spatial tree objects will store static polygon data in the leaves of the tree. Therefore, we need a data structure that we can use to pass this data into our tree (prior to the build process being called). This same structure will also be used once the tree is compiled to store the polygon data at the leaf nodes. The object we use for this is defined in CObject.h and CObject.cpp and is called CPolygon.

CPolygon is a simple class that essentially just contains a list of CVertex structures and has two methods that allow us to add vertices to its list or insert vertices in the middle of its list. We will not be showing the code to these functions as they are simple array resize and manipulation functions the likes of which we have seen many times before. However, this object also has a method called Split which allows us to pass in a plane and it will clip/split the polygon to that plane. The CPolygon object's vertex array is not altered by this process as this function allocates and returns two new CPolygon objects containing the front and back split fragments. After calling this function and retrieving the two split polygon fragments the caller will usually delete the original polygon as it is probably no longer needed. Below we show the class declaration in CObject.h.

```
class CPolygon
{
public:

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

    // Public Functions for This Class
    long            AddVertex           ( USHORT Count = 1 );
    long            InsertVertex        ( USHORT nVertexPos );
    bool            Split               ( const D3DXPLANE& Plane,
                                          CPolygon ** FrontSplit = NULL,
                                          CPolygon ** BackSplit = NULL,
                                          bool bReturnNoSplit = false ) const;

    // Public Variables for This Class
    ULONG       m_nAttribID;            // Attribute ID of face
    D3DXVECTOR3 m_vecNormal;            // The face normal.
    USHORT      m_nVertexCount;         // Number of vertices stored.
    CVertex    *m_pVertex;              // Simple vertex array
    D3DXVECTOR3 m_vecBoundsMin;    // Minimum bounding box extents of this polygon
    D3DXVECTOR3 m_vecBoundsMax;    // Maximum bounding box extents of this polygon
    ULONG       m_nAppCounter;     // Automatic 'Already Processed' functionality
    BOOL        m_bVisible;        // Should it be rendered
};
```

When our application loads static geometry from an IWF file (inside CScene::ProcessMeshes) we will store each face we load in a new CPolygon object and pass it to the spatial tree for storage. After the spatial tree's Build function has been called, the tree will contain a number of leaf structures, each containing an array of the CPolygon pointers that exist in that leaf. A CPolygon structure will contain a convex clockwise winding N-gon which means a single polygon may represent multiple triangles.

The member variables are fairly self-explanatory but we will briefly explain them here since some new ones have been added since the introduction of this class in some of our Module I lab projects.

ULONG m_nAttribID
Each polygon will store an attribute ID that has some meaning to our application (e.g., which textures and materials will need to be set when this polygon is rendered). Therefore, when we create a CPolygon object during the loading of an IWF file, we will store the global subset ID of the polygon in this member. As with our previous lab projects, this will actually be an index into the scene's attribute array where each element in that array describes the texture and material that should be bound to the device prior to rendering this polygon. Although we will not discuss our rendering strategy until the next lesson, this member is used by the tree so that it can render all polygons from all visible leaves together so that efficient batch rendering is maintained.

D3DXVECTOR3 m_vecNormal
This member will contain the normal of the polygon.

USHORT m_nVertexCount
This member contains the number of vertices currently stored in this polygon and contained in the m_pVertex array described next.

CVertex *m_pVertex
This is a pointer to the polygon's vertex array. Each element in this array is the now familiar CVertex object, which contains the positional data of the vertex, its normal, and its texture coordinates.

D3DXVECTOR3 m_vecBoundsMin
D3DXVECTOR3 m_vecBoundsMax
In these two members we will store the world space axis aligned bounding box of the polygon. That is, each polygon in our spatial tree will contain an AABB that will be used to speed up collision testing against the spatial tree's geometry database.

Our collision system will now store its static geometry in a spatial tree instead of just in a polygon array. When collision queries are performed, the collision system will ask the spatial tree for a list of all the leaves that the swept sphere's AABB intersects (broad phase). Once the collision system is returned a list of leaves that contain the potential colliders, we could just collect the polygons from each returned leaf and send them to the narrow phase process. However, the actual intersection tests performed between the swept sphere and the polygon are very expensive, and even though we have rejected a large number of polygons by only fetching polygon data from the leaves that intersect the swept sphere AABB, many of the polygons contained in these leaves may be positioned well outside this AABB. Therefore, once we have the list of leaves which the swept sphere's AABB intersects, we will loop through each polygon in those leaves and test its bounding box against the bounding box of the swept sphere. This way we avoid transforming polygons into eSpace and performing the full spectrum of intersection tests on it when we can quickly tell beforehand that the polygon and the sphere could not possibly intersect because their AABBs do not intersect. Although this might sound like a small optimization to the broad phase which would hardly seem worth the memory taken up by storing an AABB in each polygon, the speed improvements to the broad phase on our test machines were very

dramatic. Using the spatial tree and this additional broad phase step in our collision system, queries on large scenes increased in performance by a significant amount.

> **Note:** We tested our collision system using a fairly large level Quake III™ level (on a relatively low end machine) prior to adding the broad phase. Due to the fact that the collision system had to query the entire scene each frame at the polygon level, a collision query was taking somewhere in the region of 10 to 12 seconds. With the spatial tree added, the time was reduced to a few milliseconds, allowing us to achieve interactive frame rates well above 60 frames per second. What a difference, 60 frames per second versus 1 frame every 12 seconds. Even on a simple level such as colony5.iwf, frame rate jumped from 30 frames per second to over 300 frames per second with the introduction of the broad phase. Proof for sure that the broad phase component of any collision system is vitally important. Also proof that hierarchical spatial partitioning is an efficient way to get access to only the areas of the scene you are interested in for a given query.

ULONG m_nAppCounter

This member is used to avoid testing a polygon for collision multiple times if a non-clipped spatial tree is being used. As we know, if a non-clipped tree is constructed, a single polygon may span multiple leaves. This means its CPolygon pointer will be stored in the polygon array of multiple leaf structures. When the collision system fetches the list of intersecting leaves from the spatial tree it has to make sure that if a polygon exists in multiple leaves it is not tested twice.

To get around this problem we decide to add an application counter member variable to our CGameApp class. This is a simple DWORD value that can be incremented via the CGameApp::IncrementAppCounter method and retrieved via the CGameApp::GetAppCounter. In other words, this is a simple value that can be increment by external components.

Our collision system uses the app timer in the following way: Prior to a collision test, the app counter is incremented and then fetched so that we now have a new unique counter value for this update. When we fetch the intersecting leaves from the spatial tree and find a polygon that needs to be tested at the narrow phase level, we will store the app counter value in this member of the CPolygon. If a little later, when testing the polygons from another leaf, we find that we are about to test a polygon that has an m_nAppCounter value which is equal to the current value of the CGameApp's application counter, it must mean that this polygon has already been processed in this update because it belonged in a leaf that we have previously tested. Therefore, we do not need to test it again.

This is a much nicer and more efficient solution than simply storing a 'HasBeenTested' boolean in the polygon structure because this would involve us having to reset them all back to false prior to performing another collision test. We would certainly not want to have to do that for every polygon in the scene. By storing the application counter in the polygon, as soon as we wish to perform another query, we can just increment the application timer which will immediately invalidate all the polygons because their m_nAppCounter variables will no longer match the current value of the application counter.

bool m_bVisible
We will see this member of the polygon structure being used in the following lesson when we implement the rendering system. Essentially, it allows us to flag a polygon as being invisible when added to the tree. This is useful if we wish to add a polygon to the collision system but would not like to have it rendered The collision system will ignore this member and will test the swept sphere against any polygons in its vicinity. The rendering system however (which will use the same tree) will only render a polygon if it has this boolean set to true. This boolean is set to true by default in the CPolygon constructor.

Split - CPolygon

The only method of CPolygon that we have to discuss is an important one since it is the key to creating a clipped tree. The CPolygon::Split method implements the splitting strategy discussed in the last section. Unfortunately, the code may at first seem a little confusing due to two reasons. First, we want the function to also double as a clipper, so there are several conditional blocks in the code that only get executed if the caller has requested that it is interested in getting back the relevant split fragment. For example, if the caller passes NULL as the FrontSplit parameter the function will discard any portion of the polygon that lies in front of the plane and will only return the back split fragment. Second, the function is ordered in a way that means we only allocate the memory for the new polygon data when we know we actually need it.

We have also added a fourth parameter to this function, which is a boolean that allows us to specify what should happen when the passed plane does not intersect the polygon. When this boolean is set to false, the polygon data will always be created and returned in either the front or back split. That is, if the polygon is completely in front of the passed plane, a new front split polygon will be returned which contains a copy of all the data from the original polygon. This way we will always get back either a front split polygon or a back split polygon even if the polygon is not spanning the plane. Although this might sound like a strange way for the function to behave, it can be useful during a recursive clipping process to always assume that the original polygon is no longer needed after its Split method has been called. However, it is also often the case that if the polygon does not span the passed plane, then you do not want any front split or back split polygon created and would rather just continue to work with the original polygon. This is what the final parameter to this function is for. If set to true, it will only return front split and back split polygons if the original polygon is spanning the plane. If not, it will just return immediately and essentially will have done nothing. If set to false, it will always return a polygon in either the front or back split polygons even if the polygon was completely in front or completely behind the plane. In other words, the function will always create a new polygon in this instance allowing you to discard the original one.

Let us have a look at the code to this function a section at a time. There are four parameters. The first is the clip/split plane and it is expected in the form of a D3DXPLANE structure which describes the plane in AX+BY+CZ+D format. The second and third parameters are the addresses of CPolygon pointers which on function return will point to the new front and/or back polygon fragments. The caller does not have to allocate the CPolygon objects to contain the front and back splits, it simply has to pass the address of two CPolygon pointers. The Split function will allocate the CPolygon objects and assign the pointers you pass to point at them, so the caller can access them on function return. The fourth parameter

is the boolean parameter describing whether the function should only create new polygons if the polygon is spanning the plane or whether it should do nothing.

```
bool CPolygon::Split( const D3DXPLANE& Plane,
                      CPolygon ** FrontSplit,
                      CPolygon ** BackSplit,
                      bool bReturnNoSplit ) const
{
    CVertex    *FrontList     = NULL, *BackList = NULL;
    USHORT      CurrentVertex = 0, CurrentIndex = 0, i = 0;
    USHORT      InFront       = 0, Behind       = 0, OnPlane = 0;
    USHORT      FrontCounter  = 0, BackCounter  = 0;
    UCHAR      *PointLocation = NULL, Location;
    float       fDelta;

    // Bail if no fragments passed (No-Op).
    if (!FrontSplit && !BackSplit) return true;

    // Separate out plane components
    D3DXVECTOR3 vecPlaneNormal = (D3DXVECTOR3&)Plane;
    D3DXVECTOR3 vecPlanePoint  = vecPlaneNormal * -Plane.d;
```

The first section of the function is fairly simple. First, if the caller has not passed a front split or a back split pointer then we have no way of returning any clipped/split polygon data back anyway so we may as well just immediately return. It is ok to pass only one pointer (either a front split pointer or a back split pointer) which will turn the function into a clipper instead of a splitter. However, if no pointers are passed, there is no point continuing.

We can also see how we use the passed D3DXPlane structure to separate the plane into two components, a plane normal and a point on that plane. Getting the plane normal is easy as the first three members of the plane structure (a, b, and c) are the plane normal. So we can do a straight cast of these three members into a 3D vector. Calculating a point on the plane is also easy as we know that the **d** member contains the distance to the plane from the origin. That means that if we moved a point from the origin of the coordinate system in the direction of the plane normal for this distance we would have a point on that plane. As we are using the AX+BY+CZ+D version of the plane equation, D is a negative value for points behind the plane. Therefore, we just have to negate its sign and use it to scale the normal and we will have created that point on the plane.

In the next section of code we prepare the memory that will be needed for performing the split operation and the various tests. First we will need to know the location of each vertex in the polygon with respect to the plane, so we will allocate an array of unsigned chars (one for each vertex in the original polygon). In a moment, we will loop through each vertex and use the CCollision::PointClassifyPlane method we implemented earlier. This will return either CLASSIFY_FRONT, CLASSIFY_BACK or CLASSIFY_ONPLANE. We will store the results in the array.

```
    try
    {
        // Allocate temp buffers for split operation
        PointLocation = new UCHAR[m_nVertexCount];
        if (!PointLocation) throw std::bad_alloc(); // VC++ Compat
```

As you can see, this array is called PointLocation and it must be large enough to hold the classification result of each vertex in the original polygon. In a moment we will fill this array with the classification results of each vertex with respect to the plane.

What we will do now is also allocate two arrays of vertices. These two arrays will be used to collect the vertices that get added to the front split and back split polygons respectively. Of course, if the caller did not pass in either a front split or a back split pointer, then it means we do not have to allocate a vertex array to deal with vertices in that halfspace of the plane. That is, if only the address of a CPolygon pointer has been passed for the FrontSplit parameter, there is no need to allocate a vertex array to collect vertices that are in the back split. Also, if the user as passed true as the bReturnNoSplit parameter, then it means this function should only create a front split and back split polygon if the polygon is spanning the plane. Therefore, if true has been passed, we will not allocate those vertex arrays here; we will allocate them later when we know we definitely have a split case. It would be wasteful to allocate the memory for either a front split or a back split polygon only to find that we do not need it because the caller has requested that no polygons be created in the no split case.

```
        // Allocate these only if we need them
        if (FrontSplit && !bReturnNoSplit)
        {
            FrontList = new CVertex[m_nVertexCount + 1];
            if (!FrontList) throw std::bad_alloc(); // VC++ Compat
        } // End If

        if ( BackSplit && !bReturnNoSplit )
        {
            BackList = new CVertex[m_nVertexCount + 1];
            if (!BackList) throw std::bad_alloc(); // VC++ Compat
        } // End If

    } // End Try

    catch (...)
    {
        // Catch any bad allocation exceptions
        if (FrontList)      delete []FrontList;
        if (BackList)       delete []BackList;
        if (PointLocation)  delete []PointLocation;
        return false;
    } // End Catch
```

Notice that when we allocate the front and back split vertex arrays we make them large enough to hold one more than the number of vertices in the original polygon that is being clipped.

We do this because when we clip or split a polygon to a plane, although new vertices are introduced along the edges that intersect the plane (for both fragments), none of the fragments can ever have its vertex count increased beyond the original number of vertices plus one. This is shown in Figure 14.45 where a triangle in split to an arbitrary plane creating two polygons. One has four vertices and the other has three. It should be noted that this rule is only applicable to convex polygons. Since we are always dealing with convex polygons this is just fine.

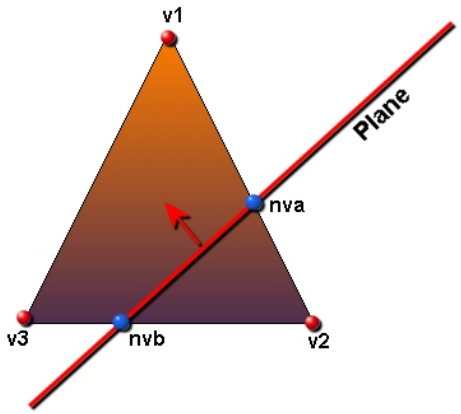


Figure 14.45

In this image we can see that the triangle to be clipped originally consisted of vertices v1, v2, and v3. However, during the testing of the edges, two were found to be spanning the plane and the vertices nva and nvb are generated and added to both the front and back vertex lists. This generates a front split vertex list of v1, nva, nvb and v3 and a back split vertex list of nva, v2 and nvb. No matter how we rotate the plane and regardless of the shape of the original polygon being clipped, we will never generate a new split polygon that has more than one additional vertex beyond the original polygon from which it was clipped/split from. So we can see in Figure 14.45 that if we allocate both the front and the back vertex lists to hold four vertices (number of vertices in the original triangle plus one) we definitely have enough room to store the vertices for each fragment.

At this point we have front and back vertex lists allocated (only if bReturnNoSplit equals false) and we also have a PointLocation array large enough to child a classification result for each vertex in the original polygon. Let us now fill out the point location array by looping through each vertex in the polygon and classifying it against the plane (using our new PointClassifyPlane function). We store the result for each vertex in the corresponding element in the PointLocation array as shown below.

```
    // Determine each points location relative to the plane.
    for ( i = 0; i < m_nVertexCount; ++i )
    {
        // Retrieve classification
        Location = CCollision::PointClassifyPlane( (D3DXVECTOR3&)m_pVertex[i],
                                                   (D3DXVECTOR3&)Plane,
                                                    Plane.d );

        // Classify the location of the point
        if (Location == CCollision::CLASSIFY_INFRONT ) InFront++;
        else
        if (Location == CCollision::CLASSIFY_BEHIND ) Behind++;
        else
        OnPlane++;

        // Store location
        PointLocation[i] = Location;

    } // Next Vertex
```

Now we have an array that stores whether each vertex is in front, behind, or on the plane. Note that we maintain three counters (InFront, Behind, or OnPlane) so that at the end of the above loop we know exactly how many vertices (if any) are in each classification pool.

In the following code section we test the value of the InFront counter and if it is not greater than zero then it means no vertices were located in the front halfspace. If the caller passed true for bReturnNoSplit, then there is nothing to do other than release the PointLocation array and return. This is because if the InFront counter is set to zero, the polygon cannot possibly span the plane. Thus in the bReturnNoSplit=true case, this means we just wish the function to return.

```
    // If there are no vertices in front of the plane
    if (!InFront )
    {
        if ( bReturnNoSplit ) { delete []PointLocation; return true; }
        if ( BackList )
        {
            memcpy(BackList, m_pVertex, m_nVertexCount * sizeof(CVertex));
            BackCounter = m_nVertexCount;
        } // End if

    } // End if none in front
```

Notice that if the InFront counter is zero but the caller did not pass true as the bReturnNoSplit parameter, all the vertices in the polygon belong to the back split fragment. When this is the case we copy all the vertex data from the source polygon into the vertex list being compiled for the back split. We also set the BackCounter variable equal to the number of vertices in the source polygon so we know how many vertices we have collected in the BackList vertex array.

In the next snippet of code we do exactly the same thing again only this time we are handling the case where no vertices were found to lie behind the plane. When this is the case we simply return if bReturnNoSplit was set to true, or we consider all the vertices of the source polygon to belong to the back split polygon and copy over its vertices into the front split vertex list.

```
    // If there are no vertices behind the plane
    if (!Behind )
    {
        if ( bReturnNoSplit ) { delete []PointLocation; return true; }
        if ( FrontList )
        {
            memcpy(FrontList, m_pVertex, m_nVertexCount * sizeof(CVertex));
            FrontCounter = m_nVertexCount;
        } // End if

    } // End if none behind
```

If both the InFront and Behind counters are set to zero then it must mean that every vertex in the source polygon lies on the split plane. This means no splitting is going to occur and the polygon does not belong in any halfspace. When this is the case we return.

```
    // All were onplane
    if (!InFront && !Behind && bReturnNoSplit )
       { delete []PointLocation; return true; }
```

Earlier in the function we allocated arrays to hold the front and back vertex lists but only if the bReturnNoSplit parameter was set to false. This is because one way or another we knew we would definitely need them even if the polygon was not spanning the plane because the function will always return a copy of the source polygon in either the front or back split polygons. However, we did not allocate these arrays if the bReturnNoSplit boolean was set to true since we would only need the front and back split vertex lists compiled if the source polygon is spanning the plane. Otherwise the function would have simply returned by now and done nothing.

In the next section of code we perform the front and back vertex list allocations for the bReturnNoSplit is true case. The memory will have already been allocated for these arrays earlier in the function if bReturnNoSplit was false.

```
    // We can allocate memory here if we wanted to return when no split occurred
    if ( bReturnNoSplit )
    {
        try
        {
            // Allocate these only if we need them
            if (FrontSplit)
            {
                FrontList = new CVertex[m_nVertexCount + 1];
                if (!FrontList) throw std::bad_alloc(); // VC++ Compat
            } // End If

            if ( BackSplit )
            {
                BackList = new CVertex[m_nVertexCount + 1];
                if (!BackList) throw std::bad_alloc(); // VC++ Compat
            } // End If

        } // End Try

        catch (...)
        {
            // Catch any bad allocation exceptions
            if (FrontList)      delete []FrontList;
            if (BackList)       delete []BackList;
            if (PointLocation)  delete []PointLocation;
            return false;
        } // End Catch

    } // End ReturnNoSplit case
```

This is all pretty familiar stuff. We know at this point that if bReturnNoSplit is true that the polygon must be spanning, otherwise we would have returned by now. So, provided the caller passed valid front and back CPolygon pointers, we allocate the vertex arrays for the front and back lists just as we did earlier for the case when bReturnNoSplit did not equal true.

At this point, regardless of the mode of the function being used, we have two empty vertex arrays that will be used to hold the vertex lists for the front and back split polygons. It is now time to look at the core piece of the function that populates the vertex lists for the front and back splits and creates new vertices when an edge is found to be spanning a plane. The code is only executed if the InFront and Behind counters are both non zero since this will only be the case when the polygon is spanning the plane. Remember, earlier in the function, if the polygon was found to be totally on one side of the plane then the source polygon's vertices were copied over into the vertex array for either the front or back split.

The following code loops through each vertex in the polygon and compares the classification result we stored for it earlier with the classification result of the next vertex in the list (the other vertex sharing the edge). If the entire edge is on one side of the plane then the vertex being tested is added to the front or back list. If however, the vertex being tested and the next vertex in the list are on opposing sides of the plane, the edge formed by those two vertices spans the plane and a new vertex is generated on the plane. This new vertex is added to both the front and the back lists.

```
    // Compute the split if there are verts both in front and behind
    if (InFront && Behind)
    {
        for ( i = 0; i < m_nVertexCount; i++)
        {
            // Store Current vertex remembering to MOD with number of vertices.
            CurrentVertex = (i+1) % m_nVertexCount;

            if (PointLocation[i] == CCollision::CLASSIFY_ONPLANE )
            {
                if (FrontList) FrontList[FrontCounter++] = m_pVertex[i];
                if (BackList)  BackList [BackCounter ++] = m_pVertex[i];
                continue; // Skip to next vertex
            } // End if On Plane

            if (PointLocation[i] == CCollision::CLASSIFY_INFRONT )
            {
                if (FrontList) FrontList[FrontCounter++] = m_pVertex[i];
            }
            else
            {
                if (BackList) BackList[BackCounter++] = m_pVertex[i];

            } // End if In front or otherwise
```

In the first section of the loop shown above we set up a loop to iterate through the PointLocation results for each vertex. Loop variable *i* contains the index of the current vertex being processed and local variable CurrentVertex contains the index of the next vertex in the list. Notice we do the modulus operation to make sure that this wraps back around to vertex zero which forms the end vertex for the final edge.

First, we test to see if the classification result for the vertex we are currently testing is CLASSIFY_ONPLANE. If it is, then the vertex itself should be added to both of the vertex lists. On plane vertices will belong to both of the split polygons as shown in Figure 14.46. You can see in the above code that when this is the case, we copy the vertex into both the front and back vertex lists. Notice

that as we do we increment the FrontCounter and BackCounter local variables that are initialized to zero at the head of the function and are incremented in this loop every time we add a vertex to a list. This is so we can keep a count of the number of vertices in each list which we will need at the bottom of the function when we allocate the back split and front split polygons.

If the current vertex is not on the plane then we test to see if it is in front of the plane and if so add the vertex to the front list. Otherwise, the vertex is behind the plane so we add it to the back list.

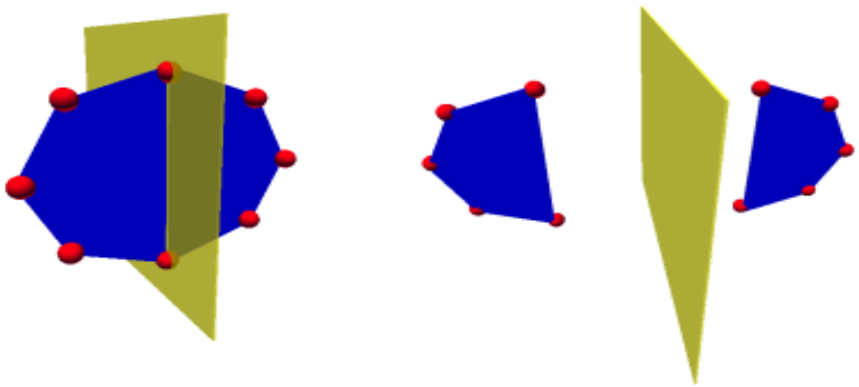


Figure 14.46 : On Plane vertices are added to both splits

We now correctly added the vertex to the front or back split but we cannot just progress to the next item in the list. If the vertex we just added and the next vertex in the list are on opposite sides of the plane we will need to treat these two vertices (the spanning edge) as a ray and intersect it with the plane.

The next section code does just that. If the next vertex in the list (CurrentVertex = i +1) is on the plane or is on the same side of the plane as the vertex we just added, then we can just continue to the next loop iteration where it will be added to its respective list.

```
        // If the next vertex is not causing us to span the plane then continue
        if (PointLocation[CurrentVertex]== CCollision::CLASSIFY_ONPLANE||
                                           PointLocation[CurrentVertex] ==
                                           PointLocation[i] )
            continue;
```

However, if we make it this far through the loop then the vertex we just added and the next vertex in the source polygon are on opposite sides of the plane and a new vertex will have to be added to both lists. This will be the vertex at the point of intersection between the edge and the plane.

We do this by making the vertex we just processed (*i*) the ray origin and then subtract the ray origin from the position of the next vertex in the list (CurrentVertex) which gives us our ray delta vector. We then use our CCollision::RayIntersectPlane function to calculate the *t* value of intersection.

```
            // Calculate the intersection point
            D3DXVECTOR3 vecOrigin  =(D3DXVECTOR3&)m_pVertex[i];
            D3DXVECTOR3 vecVelocity=(D3DXVECTOR3&)m_pVertex[CurrentVertex]
                                    - vecOrigin;
```

```
            CCollision::RayIntersectPlane( vecOrigin,
                                           vecVelocity,
                                           vecPlaneNormal,
                                           vecPlanePoint,
                                           fDelta,
                                           true );

            // create new vertex position
            CVertex NewVert;
            (D3DXVECTOR3&)NewVert = vecOrigin + (vecVelocity * fDelta);
```

When this function returns, local variable fDelta will contain the intersection point parametrically. This will be a value between 0.0 and 1.0 describing the position of intersection between the first and second vertex. You can see at the bottom of the above code that by scaling the ray delta vector (vecVelocity) by the *t* value of intersection (fDelta) and adding the result to the ray origin we have the position of the new vertex that we need to add.

Of course, we do not just need to calculate the new position of the vertex; we also have to interpolate the values of any other components that might be stored in the vertices, such as color and/or UV coordinates.

Figure 14.47 shows how a triangle polygon might be mapped to a given texture. Note the texture coordinates for v1 and v2. The red line shows where an intersection with the plane has occurred with this polygon and this is the point where a new texture coordinate needs to be created (percent = 0.5).

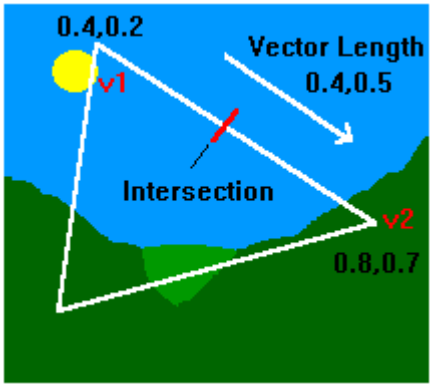


Figure 14.47

First we will subtract the first vertex texture coordinates from the second and end up with the vector length of the line between v1 and v2 in texture space. You can see that <0.8,0.7> - <0.4,0.2> = <0.4,0.5>. This is the direction and the length between texture coordinates v1 and v2. The great thing is that our RayIntersectPlane function returned a *t* value from the start of the line to the point where the intersection occurred. We can now use this value for texture coordinate interpolation, color interpolation, and even normal interpolation to generate all the other data we need for the newly created vertex. For example, imagine that the fDelta values returned from RayIntersectPlane was 0.5. This tells us that the edge intersects the plane exactly halfway between vertex v1 and vertex v2. If we multiply the texture coordinate delta vector by the *t* value (0.5 in this example) we get a vector of

<0.4*0.5, 0.5*0.5> = <0.2, 0.25>

Now we just have to add this scaled texture coordinate delta vector to the texture coordinates of the first vertex in the edge (v1) and we have the new texture coordinate of the vertex inserted at the intersection point.

New Texture Coordinates = <0.4, 0.2> + <0.2, 0.25> = <0.6, 0.45>

Incidentally, we use this same linear interpolation to generate a normal for the newly created vertex. Below we see the next section of the code that generates the normal and texture coordinate information for the new vertex.

```
            // Interpolate Texture Coordinates
            D3DXVECTOR3 Delta;
            Delta.x    = m_pVertex[CurrentVertex].tu - m_pVertex[i].tu;
            Delta.y    = m_pVertex[CurrentVertex].tv - m_pVertex[i].tv;
            NewVert.tu = m_pVertex[i].tu + ( Delta.x * fDelta );
            NewVert.tv = m_pVertex[i].tv + ( Delta.y * fDelta );

            // Interpolate normal
            Delta          = m_pVertex[CurrentVertex].Normal - m_pVertex[i].Normal;
            NewVert.Normal = m_pVertex[i].Normal + (Delta * fDelta);
            D3DXVec3Normalize( &NewVert.Normal, &NewVert.Normal );

            // Store in both lists.
            if (BackList)  BackList[BackCounter++]   = NewVert;
            if (FrontList) FrontList[FrontCounter++] = NewVert;

        } // Next Vertex

    } // End if spanning
```

Notice how the newly created vertex is added to both vertex lists. And that is the end of the loop.

At this point we have built the vertex lists for both of the polygons if applicable. Now it is time to actually allocate the new CPolygon objects (again, where applicable). In this first section we allocate the new polygon that will contain the front split vertex list. Obviously we only do this if the caller passed a valid FrontSplit pointer as a function parameter and if we have more than 2 vertices in the front list. Once the new polygon is allocated, we call its AddVertex method so that the CPolygon can reserve enough space in its vertex array for the correct number of vertices that we have compiled in the front vertex list. We then copy over the vertices that we have compiled in the temporary front list into the vertex array of CPolygon.

```
    // Allocate front face
    if (FrontCounter >= 3 && FrontSplit)
    {
        // Allocate a new polygon
        CPolygon * pPoly = new CPolygon;

        // Copy over the vertices into the new poly
        pPoly->AddVertex( FrontCounter );
        memcpy(pPoly->m_pVertex, FrontList, FrontCounter * sizeof(CVertex));

        // Copy over other details
        pPoly->m_nAttribID   = m_nAttribID;
        pPoly->m_vecNormal   = m_vecNormal;

        // Store the poly
        *FrontSplit = pPoly;

    } // End If
```

Notice in the above code that after we have copied the vertex data into the polygon, we also copy over the attribute ID and the normal of the source polygon. However much we split the polygon, every fragment will always exist on the same plane as the parent, so the normal can be safely inherited. The attribute id is also inherited into the child splits. This makes good sense because if we have a polygon with a wood texture applied and we split it, we get two polygons with that same wood texture applied. Finally, we assign the FrontSplit pointer passed by the caller to point to this newly created polygon so that when the function returns the caller will have access to it.

In the next section of code we do the same thing all over again for the back split polygon.

```
    // Allocate back face
    if (BackCounter >= 3 && BackSplit)
    {
        // Allocate a new polygon
        CPolygon * pPoly = new CPolygon;

        // Copy over the vertices into the new poly
        pPoly->AddVertex( BackCounter );
        memcpy(pPoly->m_pVertex, BackList, BackCounter * sizeof(CVertex));

        // Copy over other details
        pPoly->m_nAttribID   = m_nAttribID;
        pPoly->m_vecNormal   = m_vecNormal;

        // Store the poly
        *BackSplit = pPoly;

    } // End If
```

At this point our job is done so we release the temporary memory we used for the clipping/splitting process before returning true.

```
    // Clean up
    if (FrontList)      delete []FrontList;
    if (BackList)       delete []BackList;
    if (PointLocation)  delete []PointLocation;

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

Although that was a somewhat tricky function, the core process it performs is delightfully easy. It has been made much less reader friendly because it has many early out conditionals so that memory is not allocated unnecessarily, but nevertheless, it is still a pretty straightforward algorithm. You will see the Split method being used quite a lot during the construction of the clipped spatial tree.

We have now covered every thing we need to know about CPolygon; the structure used by our spatial trees to store static polygon data. We are now ready to look at ISpatialTree, the abstract base class for all of our new tree types and the interface our application will use to communicate with our trees.

14.11 The Abstract Base Classes

In the file ISpatialTree.h you will find the declaration of the ISpatialTree class. This is an abstract interface which exists to provide a consistent means of communication between our application and our various trees. As long as our quad-tree, oct-tree and kD-tree classes are derived from this interface and implement its methods, our application can use the same code to interface with any of these tree types. All the trees we have discussed share a lot of common functionality. For example, regardless of how they are built, they all contain an array of leaf nodes and an array of CPolygons assigned to each leaf node.

The ISpatialTree interface exposes all the methods an application will need to communicate with a spatial tree. We can think of it as defining a minimum set of functionality that must be implemented in our derived classes. It enforces for example that each of the trees we implement has an AddPolygon method so that an application can add CPolygons to its static geometry data. It enforces that our derived classes must implement an AddDetailArea method so that the application can register areas of interest for use during the build process. Another example of a method that must be implemented is the Build method which an application can use to instruct the spatial tree to compile its spatial information using the polygon data and detail areas that have been assigned to it. Further, in the ISpatialTree.h file, you will also see several structures and typedefs defined that will be used by all of our derived classes. Let us start to have a look at what is inside this file now.

The first structure defined in this file is the one that is used to pass detail information to and from the tree. This structure is very simple and has three members. The first two should contain the minimum and maximum extents of its world space AABB and the second is a context pointer that can be used by the application to point at any data it so chooses. Beyond the examples discussed earlier, this context pointer might point to some structure that contains settings to configure fog on the device. The detail area in this instance would be used to represent an area within the scene where fog should be enabled during rendering. The TreeDetailArea structure is shown below and is something we will see being used in a moment.

Excerpt From ISpatialTree.h

```
typedef struct _TreeDetailArea
{
    D3DXVECTOR3     BoundsMin;      // Minimum AABB extents of the area
    D3DXVECTOR3     BoundsMax;      // Maximum AABB extents of the area
    void          * pContext;       // A context pointer that can be assigned.

} TreeDetailArea;
```

Because all our various tree types will have support for the registration of detail areas, we have defined this structure in the main header file and have made the AddDetailArea and GetDetailAreaList methods members of the base class.

14.11.1 ILeaf – The Base Class for all Leaf Objects

Another thing that all of our trees will have in common is that they will all need the ability to store leaf information at terminal nodes in the tree. There are certain methods that we will want a leaf object to expose so that the tree (or application) can interface with a leaf more easily. For example, we will usually want a leaf to have an IsVisible method so that the application can query whether a given leaf is visible and should have its contents rendered. We will also want the ability to get back a list of all the detail areas and polygons that are assigned to a leaf. Furthermore, we know that each leaf will also need to store an AABB describing the region of space it occupies. Obviously, the leaf is only visible if this bounding box is contained either fully or partially inside the camera frustum.

Although we may want a leaf to contain much more information than this, the abstract ILeaf class is shown below and defines the base interface that our application will use to access and query a leaf. As with the ISpatialTree class, it is abstract, so it can never be instantiated directly. It must be derived from so that the function bodies can be implemented. These base interfaces force us to support and implement all the function in the base class when writing any tree type now or in the future. If we do not, we will get a compiler error. As long as we implement the functions specified in the base classes our application will happily work with the tree as its spatial manager with no changes to the code. This is because our application will only ever work with the ISpatialTree and ILeaf. As long as we derive our classes from these interfaces and implement the specified methods, we are in good shape to plug in whatever tree types we like down the road.

Below we see the class declaration for ILeaf.

```
class ILeaf
{
public:
    // Constructors & Destructors for This Class.
    virtual ~ILeaf() {};  // No implementation, but forces all 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 void            GetBoundingBox      ( D3DXVECTOR3 & Min,
                                                  D3DXVECTOR3 & Max ) const = 0;
};
```

All of the methods shown above must be implemented in any derived leaf objects we create since they will be used by the spatial tree. Let us first discuss what these methods are supposed to be used for and what information they should return.

virtual bool IsVisible ( ) const = 0;
It is very important that a leaf expose this method as it is the application's means of testing whether a given leaf was found to be inside the frustum during the last update. This is very useful for example when performing queries on the tree such as testing to see if a dynamic object should be rendered.

virtual unsigned long GetPolygonCount ( ) const = 0;
This method returns the number of polygons contained in the leaf. This is very important and will be used often by our collision system. For example, whenever the player's position is updated, an AABB will be constructed and fed into the spatial tree's CollectLeavesAABB method. This method will traverse the tree and compile a list of all the leaves the passed AABB was found to intersect. In the case our broad phase collision pass, this will allow us to get back a list of leaf pointers after having passed the swept sphere's bounding box down the tree. We know that only the polygons in those leaves need to tested in the narrow phase. Of course, in order to test each polygon in the leaf we must set up a loop to count through and retrieve these polygons from the leaf. This method would be used to retrieve the number of polygons in the leaf that will need to be extracted and passed to the narrow phase.

virtual CPolygon * GetPolygon (unsigned long nIndex) = 0;
This method is used by the collision system (for example) to retrieve the CPolygon object in the leaf's polygon array at the index specified by the parameter. For example, if the GetPolygonCount function returns 15, this means that you will have to call GetPolygon 15 times (once for each index between 0 and 14) to extract each polygon pointer.

virtual unsigned long GetDetailAreaCount ( ) const = 0;
Each leaf may also contain an array of detail areas. That is, when the tree is built, any detail areas that had been registered with the tree prior to the build process commencing will have their pointers copied into a leaf's DetailArea array for any leaf whose bounding box intersects the bounding box of the detail area. A detail area will typically span many leaves, so it is quite typical during the build process for the same detail area to have its pointer stored in the DetailArea array of multiple leaf nodes.

virtual TreeDetailArea *GetDetailArea (unsigned long nIndex) = 0;
This method allows the application to fetch a detail area from the leaf. The input index must be between 0 and one less than the count returned from GetDetailAreaCount method.

virtual void GetBoundingBox(D3DXVECTOR3 & Min, D3DXVECTOR3 & Max) const = 0;
It can prove handy to be able to retrieve the AABB of a leaf, so this function provides that service. When passed two vectors, it will fill them with the minimum and maximum extents of the leaf's AABB.

14.11.2 ISpatialTree – The Base Tree Class

The ISpatialTree class is the abstract base class from which our spatial trees will be derived. It specifies the core functions that our derived classes must implement in order to facilitate communication with the application. Provided our derived classes (quad-tree, kD-tree, oct-tree, etc.) are all derived from this class and implement all the pure virtual functions specified, our application can work with all our tree objects using this common interface. This means that our application code will never change even if we decide to switch from a quad-tree to an oct-tree. The class also defines some types that the derived classes can use to declare polygon lists, leaf lists, and detail area lists. ISpatialTree contains no member variables and no default functionality, so we can never instantiate an object of this type. However, it does provide a consistent API that our derived tree classes must support in order for our application to effortlessly switch between tree types and build and query the given tree. How these functions are actually implemented in the derived classes is of no concern to the application, just so long as they perform the desired task.

ISpatialTree is contained in the file ISpatialTree.h, along with ILeaf, and its declaration is shown below.

```
class ISpatialTree
{
public:

    // Typedefs, Structures and Enumerators.
    typedef std::list<ILeaf*>           LeafList;
    typedef std::list<CPolygon*>        PolygonList;
    typedef std::list<TreeDetailArea*>  DetailAreaList;

    // Constructors & Destructors for This Class.
    virtual ~ISpatialTree() {};     // No implementation, but forces all derived
                                       classes to have virtual destructors

    // Public Pure Virtual Functions for This Class.
    virtual bool            AddPolygon       ( CPolygon * pPolygon ) = 0;
    virtual bool            AddDetailArea    (const TreeDetailArea & DetailArea)=0;
    virtual bool            Build               ( bool bAllowSplits = true ) = 0;
    virtual void            ProcessVisibility   ( CCamera & Camera ) = 0;
    virtual PolygonList    &GetPolygonList      ( ) = 0;
    virtual DetailAreaList &GetDetailAreaList   ( ) = 0;
    virtual LeafList       &GetLeafList         ( ) = 0;
    virtual bool            GetSceneBounds      ( D3DXVECTOR3 & Min,
                                                  D3DXVECTOR3 & Max ) = 0;
    virtual bool            CollectLeavesAABB   ( LeafList & List,
                                                  const D3DXVECTOR3 & Min,
                                                  const D3DXVECTOR3 & Max ) = 0;
    virtual bool            CollectLeavesRay    ( LeafList & List,
                                                  const D3DXVECTOR3 &RayOrigin,
                                                  const D3DXVECTOR3 &Velocity)= 0;

    // Public Optional Virtual Functions for This Class.
    virtual bool            Repair              ( ) { return true; }
    virtual void            DebugDraw           ( CCamera & Camera ) {};
    virtual void            DrawSubset          ( unsigned long nAttribID ) {};
};
```

The class contains a lot of function declarations and some typedefs which define how the system should work. Before moving on and examining how these methods are implemented in the derived class, let us first discuss what each method is expected to do so that we understand what is required of us in our derived class implementations.

typedef std::list<ILeaf*> LeafList
typedef std::list<CPolygon*> PolygonList
typedef std::list<TreeDetailArea*> DetailAreaList

These three type definitions are conveniences that the derived class can use to allocate variables of a certain type. For example, all of our trees will maintain a list of CPolygon pointers. Although you can feel free to store these polygon pointers in any format you choose in your derived class, the base class defines some handy type definitions allowing us to easily declare an STL list of polygons using the variable type 'PolygonList'. You can also see that there are type definitions that describe variables of type LeafList and DetailAreaList to be STL lists of those respective structures. Remember, these are just type definitions, so this class has no members and you can feel free not to use these types to store your polygon, leaf, and detail area data. However, we use variables of the types shown above in our derived classes to store the three key data elements managed by the tree (polygons, leaves, and detail areas).

virtual bool AddPolygon (CPolygon * pPolygon)
virtual bool AddDetailArea (const TreeDetailArea & DetailArea)

It makes sense that all of our derived tree classes will need to implement a function that will allow the application to add polygon and detail area data to its internal lists prior to its Build function being called. The two functions shown above must be implemented in the derived class so that the application has a means to do that. For example, when the application loads a static polygon from an IWF file, it will store it in a CPolygon structure and call the tree's AddPolygon method to register that polygon with the tree. This method will be implemented such that it stores the passed polygon pointer in its internal polygon pointer list (a variable of type PolygonList). The AddDetailArea is expected to be implemented in the same way so that the application can register areas of interest with the tree class. They will be used later in the build process to include areas that contain no polygon data (for example) within the volume of space partitioned by the tree.

virtual bool Build (bool bAllowSplits = true)

Every tree class that we develop will certainly need to expose a build function that allows the application to instruct the object to build its spatial hierarchy once it has registered all the polygon data and detail area data with the object. The build function of the derived class will be implemented differently depending on the type of tree being built. When the build function returns control back to the application, the spatial hierarchy will have been constructed and it will be ready for collision querying and visibility testing.

All of our trees will also have the option during creation to build a clipped on non-clipped tree. Thus, conditional logic will need to be put in place to determine what action should be taken if a polygon spans the bounds of a leaf in which it is partially contained. The bAllowSplits parameter to this function is the application's means of letting the tree know which type of build strategy should be employed. If the parameter is set to true, a clipped tree will be built. Any polygons that span multiple leaf nodes will be clipped to the bounds of each leaf in which it is contained and every leaf will ultimately contain a list

of polygons that are contained totally within its bounds. If this parameter is set to false, then any polygon that spans multiple leaf nodes will have its pointer stored in each of those leaf nodes.

> **Note:** Remember that clipping the polygons to the leaf nodes increases the polygon count of the level, potentially by a considerable amount. This means more polygons need to be rendered and more draw primitive calls have to be made. In our test level, this was a major problem with the oct-tree, where the polygon count increased between 60% and 90% when clipping was being used (depending on leaf size).

virtual void ProcessVisibility (CCamera & Camera)
Our application will also expect every tree class to implement the ProcessVisibility method, which exists to perform a hierarchy traversal with the passed camera and set any leaves that exist inside or partially inside the frustum as visible. This function, along with much of the rendering system of our tree classes, will be discussed in detail in the next chapter since this chapter is going to focus on the core building code. For now, just know that this function will be called by the scene prior to the tree being rendered. When this function returns, each leaf in the tree will know whether it is visible or not. When the application then issues a call to the ISpatialTree::DrawSubset method, the tree will know that it only has to render polygons that are contained in leaves that currently have their visible status set to true. Obviously, this will allow us to reject a lot of the geometry from being rendered most of the time.

virtual PolygonList &GetPolygonList ( )
virtual DetailAreaList &GetDetailAreaList ( )
virtual LeafList &GetLeafList ( )
These three methods must be implemented to allow the application to retrieve the polygon list, the detail area list and the list of leaves being used by the tree. This might be useful is the application would like to save the compiled tree data to file or would like to perform some custom optimization on the data. The GetPolygonList method is essential, since after the build process has completed, the internal list of polygons used by the tree might be quite different from the list the application originally registered with the tree. If clipping is being used, many of the polygons from the original list added by the application will have been deleted and replaced by the fragments they were divided into. Therefore, these functions are usually called after the build process so that the application has some way to access the data stored in the tree. Of course, the leaf list is not even compiled until the tree is built, so it would serve no purpose to call this function prior to calling the Build function.

virtual bool GetSceneBounds (D3DXVECTOR3 & Min, D3DXVECTOR3 & Max)
This method allows the application to retrieve an axis aligned bounding box that bounds the entire area of space partitioned by the tree. This is essentially the bounding box of the root node of the tree. Of course, this is not necessarily an AABB bounding only the static polygon data, it will also account for detail areas. Our derived classes will implement this method by simply returning the bounding volume of the root node.

virtual bool CollectLeavesAABB (LeafList & List, const D3DXVECTOR3 & Min, const D3DXVECTOR3 & Max)
This key method is one that the collision system uses to run queries on the tree. When this function is called, the application will pass an axis aligned bounding box and an empty leaf list (an STL list of ILeaf structures). This function should be implemented such that it will traverse the tree from the root node

and collect the leaves whose volumes are intersected by the passed AABB, attaching them to the input list. We can think of two situations where this function will be useful immediately.

When implementing the broad phase of our collision system we will wish to pass only polygons in the immediate area of the swept sphere's AABB to the narrow phase. In order to do this we will want to pass the swept sphere's AABB into the tree and get back a list of leaves that intersect it. It is only the polygons in those leaves that will have to be further checked at a lower level. Any polygons not contained in those leaves will never be considered by the collision system. This function will speed up our collision system by an order of magnitude.

Another time when our application will need to use this method is when determining in which leaves a dynamic object is located. Whenever a dynamic object has its position updated, we can pass its AABB into the tree and get back a list of intersected leaves. These are the leaves the dynamic object is currently contained within. This leaf list will be maintained internally for the object, but we can retrieve it via an ID prior to executing any draw calls. Before the scene renders any geometry, it will call the ISpatialTree::ProcessVisibility method which will determine which leaves are visible and which leaves are not. Since we know what leaves a dynamic object resides in, we can quickly test whether any of them were visible and take appropriate measures.

This function will be implemented as a simple recursive procedure that traverses the tree from the root node and performs and AABB/AABB intersection test between the passed AABB and the bounding volume of the node currently being visited. Once again, this is very fast because as soon as we find a node whose volume does not intersect the query volume, we never have to bother stepping into its child nodes, thus rejecting huge portions of the tree from having to be tested.

This function will need to be implemented differently in each derived class because the layout of the node structure and the way the tree is traversed will differ between tree types. For example, in a quad-tree we will need to step into four children at each node, while the oct-tree would recur into eight children. Of course, the underlying algorithm will be the same: perform AABB/AABB tests at each node and determining whether traversing into the children is necessary. Whenever we traverse into a leaf, it means that leaf is contained inside the query volume and it is added to the passed leaf list so that the caller will have access to it. We will discuss how to perform AABB/AABB intersections tests in a moment when we add that functionality to our CCollision class.

virtual bool CollectLeavesRay (LeafList & List, const D3DXVECTOR3 &RayOrigin,
const D3DXVECTOR3 &Velocity)

This function has an almost identical purpose to the previously described function, only this time the query object is a ray instead of an AABB. There will be several times when we may wish to determine which polygons in the scene a ray intersects, and this function will allow us to do this efficiently.

The function is passed the ray origin and delta vectors along with an empty leaf list. The function should be implemented such that it will traverse the tree and return a list of leaves whose bounding volumes were intersected by the ray. Once again, this will have to be implemented slightly differently in each derived class as the way in which the tree is traversed is dependant on the node type (i.e., different numbers of children).

The function will step through each node in the tree and perform a ray/AABB intersection test with the node's bounding volume. If the ray does not intersect a volume then its child nodes do not have to be tested. For any node whose bounding box intersects the ray, we step into its children and perform the same process. Whenever we reach a leaf node, it means the ray must be partially contained within that leaf and the leaf structure is added to the leaf list so it can be returned to the caller.

virtual bool Repair ( ) { return true; }

The repair function is not an abstract function as it does have an implementation in ISpatialTree that essentially does nothing. This means that we do not have to bother implementing this function in the derived classes. However, it does give us an opportunity to perform some optimizations on the tree polygon data after the tree has been built. We will implement this function in our derived classes to perform a weld operation on the polygon vertices allowing to get rid of any redundant vertices (i.e., vertices that share the same position in 3D space and have the same attributes). This allows us to cut down on the number of vertices used by our scene quite dramatically in certain cases. In the case of GILES™, every face will have its unique vertices, so a cube constructed from 6 faces will actually contain 6*4=24 vertices. If the attributes of each face are identical, then at each corner of the cube there will be three vertices sharing the same space that have the same attributes (1 for each face that forms that corner). By performing a weld operation, we can collapse these three vertices at each corner into a single vertex that each of the three faces will index.

Another task that we will perform inside our Repair function will be the repair of T-junctions. T-junctions will be explained later, but as mentioned earlier, they frequently occur when lots of clipping has been performed. Since they cause very unsightly visual artefacts, we will definitely wish to repair them if we have built a clipped tree and intend to use the tree for rendering. Even if a non-clipped tree is being used, it is not uncommon for the source geometry to contain T-junctions that the artist may have inadvertently created, so we can repair those too.

If you do decide to derive your own tree types from ISpatialTree, you do not have to implement this step. Everything will still work fine because the default implementation will do nothing and just return. This method will be called by the scene after the Build function has returned and the tree has been completely built.

virtual void DebugDraw (CCamera & Camera) {}

This function is another function that has a default implementation that does nothing, so you do not have to implement it in your derived classes. However, it is often very useful for the application to be able to get some debug information about how the tree was constructed. As you have no doubt become aware, finding potential problems in recursive code can be very difficult, so if your spatial tree is not behaving in quite the way it should, it is useful to have this function available so that some rendering can be done to help you visualize how space has been partitioned. It is also useful when you are trying to configure the settings for your scene, such as the minimum leaf size (i.e., the size at which we make a node a leaf regardless of how much data it contains). As the leaf size will be directly related to the scale of your scene, you will likely want a way to see how the space has been partitioned.

In each of our derived tree classes we implement this method so that we can render the bounding boxes around each leaf node. Figure 14.51 shows a screen shots from Lab Project 14.1 with debug drawing enabled for a kD-tree (available via a menu option). On the code side, all the CScene::Render method

does after rendering the tree polygons and any dynamic objects, is call the tree's DebugDraw method. This method traverses the tree searching for visible leaf nodes. Once a leaf node is found it fetches the bounding box of the leaf and draws it using a line list primitive type.

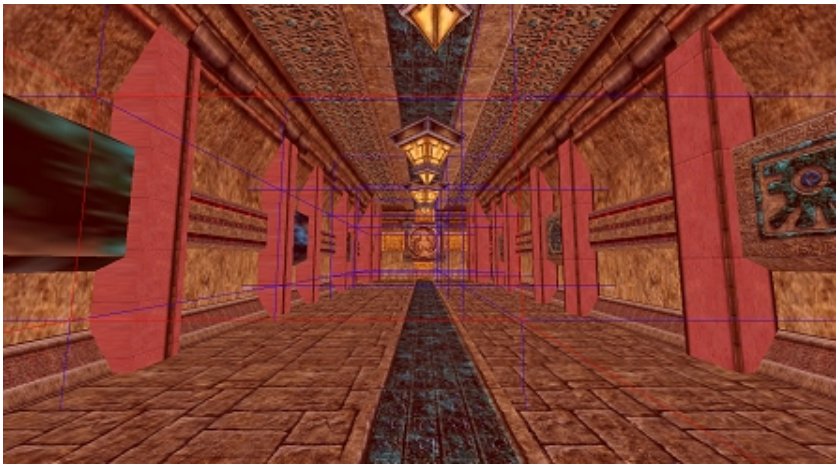


Figure 14.51

With debug draw enabled we can easily see the size of each leaf node and how the space is partitioned which, as mentioned, is very useful indeed when diagnosing various issues. In Figure 14.51 you can see that the bounding box of the leaf in which the camera is currently located is rendered in red instead of blue like all other leaf nodes. When the debug draw renders a red box it means that you are currently located in a leaf that contains something; either geometry or a detail area. However, if the leaf in which the camera is currently located is an empty leaf node the box is rendered in green as shown in Figure 14.52.

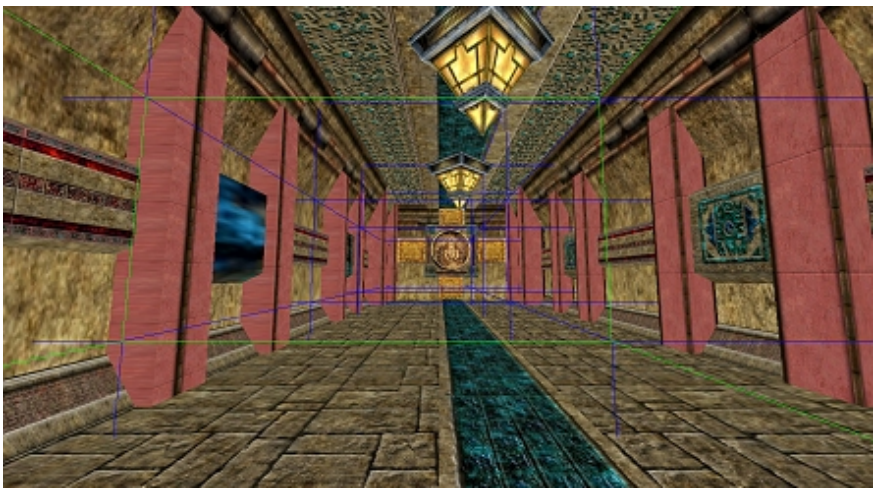


Figure 14.52

The DebugDraw method is a method that will have to be implemented slightly differently in each derived tree class. Although the output will be the same in all versions of this function (for the quad-tree, oct-tree and kD-tree) it is a function that needs to traverse the tree hierarchy and therefore, is dependant on the node structure of the tree being used and the number of children spawned from each node. However, each version of this function (for each tree type) will be almost identical. They will all traverse the tree looking for leaf nodes and then render bounding boxes using the bounding volume information. We will look at the code to these functions later in the lesson.

virtual void DrawSubset (unsigned long nAttribID) {}

In theory, rendering a spatial tree is very easy. For example, we could just traverse the hierarchy and collect the polygons from the visible leaf nodes and copy them into dynamic index and vertex buffers. Unfortunately, with such a scheme, memory copying really hampers performance when we are building those render buffers. Using a dynamic index buffer only approach is certainly much better, but still not optimal. Rendering a tree efficiently in a way that does not hamper the performance of powerful 3D

graphics cards is not so easy and in the next chapter we will discuss the rendering system that we have put in place that all of our spatial trees will use to collect and render their geometry.

> **Note:** Since the spatial tree rendering system will be rather involved, we will dedicate a good portion of the next chapter entirely to this one topic. Although this rendering system is used in Lab Project 14.1, note that this same lab project will be discussed over these two chapters. In this chapter we will discuss building the spatial tree and the upgrade of the collision system to use the spatial tree during its broad phase pass. In the next lesson we will discuss exactly how the tree data is converted into hardware friendly render batches so that we can minimize draw primitive calls and maximize batch rendering. When examining the code to Lab Project 14.1, it may be best for you to ignore how the tree is rendered for now.

The DrawSubset function will need to be implemented by each tree class so that the application can instruct the tree that it needs to render all visible polygons belonging to the passed subset. As we know, our scene is responsible for setting textures and materials and it does not want to set the same one more than once if it can be avoided. That is, we wish to setup a given attribute and then render all the polygons in the scene that use it. This minimizes render state changes and DrawPrimtive calls, which can really help performance. The CScene::Render method will first call the ISpatialTree::ProcessVisibility method prior to any rendering taking place. This instructs the tree to traverse its hierarchy and mark any leaves that are inside the view frustum as being visible. Our scene will then loop through each attribute used by the scene and call this method, each time passing in the current attribute/subset being processed. This will instruct the tree to render any polygon in any of the currently visible leaves that belong to this subset. As you will see in the next lesson, doing this efficiently means putting quite a complex system in place.

Although we have not yet discussed the implementation details of any of the functions declared in the base class, discussing ISpatialTree and the methods it exposes has given us an understanding of how our tree will work, the functions that we are expected to implement in our derived classes, and the way in which our application will communicate with our trees both when querying it and rendering it. With that said, let us start coding.

14.11.3 CBaseTree & CBaseLeaf – Base Functionality Classes

For the most part, in this course we have not really had to implement chains of derived classes. At most we have sometimes implemented a base class and a derived class as this makes the code clearer to read, easier to learn, and it makes program flow easier to follow (and of course, it is often good OOP). However, sometimes it makes sense to have an additional layer of inheritance when all your derived classes will share many properties and would need to have identical code duplicated for each. For example, regardless of the tree types we implement, they will all share a lot of common ideas. Each tree type will store a list of polygons, leaves, and detail areas, and will need to implement methods that allow the application to add and retrieve elements to/from these lists.

Likewise, the leaf structure used by each tree will essentially be the same and will have the same tasks to perform. Each will store a bounding box and will need to implement methods to add polygons and detail areas to its internal lists and expose functions which allow the tree to set its visible status during a

visibility update. Furthermore, although we will not discuss the rendering subsystem in this chapter, the same system will be used by each of the derived tree classes. If we were simply going to derive our quad-tree, kD-tree, and oct-tree classes directly from ISpatialTree, we would have to implement this code in each of our derived classes. This is wasteful since we would be duplicating identical code in each derived class. For example, the AddPolygon method will be implemented identically in each tree type; it will be a simple function that accepts a CPolygon pointer and stores that pointer in the tree's polygon list.

To minimize redundancy across these common tasks, we will include a middle layer called CBaseTree. CBaseTree will be derived from ISpatialTree and will implement many of the functions that are common to all tree types. It will also implement all the code to the rendering system which will be used by each derived tree class. Although we will never be able to instantiate an object of type CBaseTree (as it does not implement all the method from ISpatialTree) it will provide all of the housekeeping code. We will derive our quad-tree, oct-tree, and kD-tree classes from CBaseTree.

Due to the fact that CBaseTree contains a lot of code that is common to all tree types, implementing our actual tree classes will involve just the implementation of a few functions. For example, when implementing a quad-tree class, we essentially just have to implement the methods from ISpatialTree that are not implemented in CBaseTree. One such method is the Build method which will obviously be different for each tree type. The only other method we will have to implement in the lower level tree classes are the query routines such as CollectLeavesAABB, CollectLeavesRay, and ProcessVisibility. As these methods are tree traversal methods, they must be implemented by the actual types. Thus, if you open up the CQuadTree.cpp file for example, you will see that very little code is contained in there, as most of the common functionality is contained inside CBaseTree.

In this next section we will discuss the implementation of the CBaseTree methods that are used during the building phase. The rendering system contained in CBaseTree will be discussed in the following lesson, so its methods will be removed from the class declaration at this time. In this lesson, we are just concentrating on the CBaseTree methods used by the tree building process and the methods associated with querying the tree (such as the method used by our collision system). The code to CBaseTree and CBaseLeaf are stored in the files CBaseTree.h and CBaseTree.cpp.

14.11.4 CBaseLeaf – The Source Code

CBaseLeaf is derived from ILeaf and implements all the functionality of the base class. That is, all of our trees will store leaves of type CBaseLeaf. The class declaration is shown below and we will discuss it afterwards. Notice how the first pool of function declarations are those from ILeaf that will be implemented in this class. This is followed by some functions which CBaseTree will need to communicate with a leaf.

```
class CBaseLeaf : public ILeaf
{
public:

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

```
            CBaseLeaf( CBaseTree *pTree );

    // Public Virtual Functions for This Class (from ISpatialTree).
    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 );

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 polygon pointers in this leaf.
    DetailAreaVector    m_DetailAreas;  // Array of detail area pointers in leaf.
    bool                m_bVisible;     // Is this leaf visible or not?
    D3DXVECTOR3         m_vecBoundsMin; // Minimum bounding box extents
    D3DXVECTOR3         m_vecBoundsMax; // Maximum bounding box extents.
    CBaseTree          *m_pTree;        // The tree that owns this leaf
};
```

Looking at the above class declaration we can see that it implements those methods from the base class that allows the application to perform queries on the leaf such as retrieving the leaf's visible status or retrieving its bounding volume. Following these declarations are the functions that are new to this class which CBaseTree and any class derived from it can use to add detail areas or polygons to the leaf's arrays. It also exposes methods which CBaseTree (or any class derived from it) can use to set the visibility status of a leaf (inside the ProcessVisibility function) and methods allowing the tree building functions to set the leaf's AABB.

Following that are two type definitions called PolygonVector and DetailAreaVector. These are STL vectors which the leaf will use to store the polygon pointers and detail area pointers assigned to it. Unlike the CBaseTree which stores its polygon and detail area data in STL lists (linked lists) for efficient manipulation during the tree building process, the data that ends up being assigned to a leaf remains static once the leaf has been built. Therefore, we use vectors for faster access.

Finally, at the bottom of the declaration we can see the member variables that each leaf structure contains. They are discussed below, although their meaning will most likely be self-explanatory given their names.

PolygonVector m_Polygons
This member is an STL vector that will be used to store the polygons assigned to this leaf. During the build process, once a node has been reached which suits the criteria for being a terminal node (such as its bounding volume is sufficiently small or the number of polygons is below a certain threshold) a new CBaseLeaf object will be created and added to the tree's leaf list. The leaf structure will also be attached to the node in the tree and the polygon data that made it into that node will be added to the leaf via the CBaseLeaf::AddPolygon method. This method will store the passed CPolygon pointer in this vector.

DetailAreaVector m_DetailAreas
In an almost identical manner to the method described above, this vector will be used to store pointers to any detail objects that exist inside or partially inside the leaf.

bool m_bVisible
This member is used internally by the leaf to record its current visibility status. If, during the last ProcessVisibility test, its bounding box was found to be inside or partially inside the frustum, the tree would have set this boolean to true through the use of the CBaseLeaf::SetVisible method. The application (or the tree itself) can query the visibility status of a leaf using the CBaseLeaf::IsVisible method, which simply returns the value of this boolean.

D3DXVECTOR3 m_vecBoundsMin
D3DXVECTOR3 m_vecBoundsMax
Each leaf will store a bounding volume (an AABB) which will be set by the tree during the build process. When a leaf is created, the bounding box of the polygon data and the detail area data that made it into that node is computed. The tree will then use the CBaseLeaf::SetBoundingBox method to set the bounding box for the leaf.

CBaseTree * m_pTree
Each leaf will store a copy of the pointer to the tree which owns it. We will see how this pointer is used in the next chapter when we discuss rendering.

Let us now take a look at the method implementations shown above. Remember, if you are following along with the lab project source code files open, you will see many other method in CBaseTree that we have not discussed here. These are methods related to the rendering system which will be fully explained in the next lesson.

Constructor - CBaseLeaf

The CBaseLeaf constructor could not be simpler. We just initialize its visibility status to true. We will assume that the default state of any node/leaf in the tree is visible so that if for some reason we do not wish to perform the ProcessVisibility pass, all leaves will be rendered.

```
CBaseLeaf::CBaseLeaf( CBaseTree *pTree )
{
    // Reset required variables
    m_bVisible          = true;
```

```
     .. render data ..

    // Store the tree
    m_pTree = pTree
}
```

When the leaf is first created its polygon and detail area vectors will be empty and its bounding box will be uninitialized. Note that the leaf also accepts and stores a pointer to the base tree to which it belongs.

Setting/Getting the Leaf's Visibility Status - CBaseLeaf

The method that allows the application to query the visibility status of a leaf with a simple one line function that returns the value of the leaf's visibility flag.

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

Likewise, the method that allows our tree classes to set the visibility status of a leaf (during the ProcessVisibility pass) is a one line function that sets the boolean member equal to the boolean parameter passed.

```
void CBaseLeaf::SetVisible( bool bVisible )
{
    // Flag this as visible
    m_bVisible = bVisible;
}
```

AddPolygon – CBaseLeaf

The AddPolygon method is called by the tree building process whenever a terminal node is encountered. A new CBaseLeaf object is allocated and its pointer is stored in the terminal node. The AddPolygon method will then be called for each polygon that made it into the terminal node. The method adds the passed CPolygon polygon pointer to the leaf's polygon vector (with exception handling to return false should an error occur in the process).

```
bool CBaseLeaf::AddPolygon( CPolygon * pPolygon )
{
    try
    {
        // Add to the polygon list
        m_Polygons.push_back( pPolygon );

    }
```

```
    catch ( ... )
    {
        return false;

    }

    // Success!
    return true;
}
```

AddDetailArea – CBaseLeaf

When a leaf is created and added to a terminal node during the building process, any detail area that made it into that node will also be added to the leaf. The AddDetailArea adds the passed TreeDetailArea pointer to the leaf's internal detail area vector. The function code is shown below.

```
bool CBaseLeaf::AddDetailArea( TreeDetailArea * pDetailArea )
{
    try
    {
        // Add to the detail area list
        m_DetailAreas.push_back( pDetailArea );

    } // End Try Block

    catch ( ... )
    {
        return false;

    } // End Catch Block

    // Success!
    return true;
}
```

Retrieving the Polygon Data from a Leaf – CBaseLeaf

It will often be necessary for the application to retrieve the polygon data stored in a leaf. This is certainly true in our broad phase collision step when we will send the swept sphere's AABB down the tree and get back a list of intersecting leaves using the tree's CollectLeavesAABB method. Once this list of leaves is returned, the broad phase can fetch each polygon from each returned leaf and test it more thoroughly (first with an AABB/AABB test between the AABB of the swept sphere and the AABB of the polygon and then with the more expensive narrow phase if the prior test returns true for an intersection).

For the collision system to get this information, it must know how many polygons are stored in a leaf and have a way to access each polygon in that leaf. Below we see the implementations of the

CBaseLeaf::GetPolygonCount and CBaseLeaf::GetPolygon methods which are part of the ISpatialTree interface. The GetPolygonCount method simply returns the size of the leaf's internal CPolygon vector.

```
unsigned long CBaseLeaf::GetPolygonCount( ) const
{
    // Return number of polygons stored in our internal vector
    return m_Polygons.size();
}
```

The GetPolygon method accepts an index (in the range of zero to the value returned from GetPolygonCount - 1) and returns the CPolygon pointer stored at that location in the vector.

```
CPolygon * CBaseLeaf::GetPolygon( unsigned long nIndex )
{
    // Validate the index
    if ( nIndex >= m_Polygons.size() ) return NULL;

    // Return the actual pointer
    return m_Polygons[nIndex];
}
```

Retrieving the Detail Area Data from a Leaf – CBaseLeaf

An application may also wish to retrieve the information about which detail areas are stored in a leaf. For example, perhaps you have inserted a specific detail area that has a context pointer that points to a structure filled with fog settings. Whenever the camera is in a leaf which contains such a detail area, the pipeline's fog parameters could be set and enabled as described by the detail area's context pointer. Such a task could be performed by finding the leaf the camera is currently in and then fetching the number of detail area areas assigned to this leaf. You could then set up a loop to extract and test each detail area. If a detail area is found which has a context pointer that points to a fog structure, fog could be enabled.

In order to do this we would need to be able to fetch the number of detail areas in a leaf and expose a means for those detail areas to be retrieved and examined. The CBaseLeaf::GetDetailAreaCount is a simple function that returns the size of the leaf's detail area vector:

```
unsigned long CBaseLeaf::GetDetailAreaCount( ) const
{
    // Return number of polygons stored in our internal vector
    return m_DetailAreas.size();
}
```

The CBaseLeaf::GetDetailArea method is passed an index between 0 and the value returned by the GetDetailAreaCount function minus 1. It returns the TreeDetailArea pointer stored at that position in the vector.

```
TreeDetailArea * CBaseLeaf::GetDetailArea( unsigned long nIndex )
{
    // Validate the index
```

```
    if ( nIndex >= m_DetailAreas.size() ) return NULL;

    // Return the actual pointer
    return m_DetailAreas[nIndex];
}
```

Setting and Retrieving a Leaf's Bounding Box - CBaseLeaf

The application may want to retrieve the leaf bounding box for custom intersection routines. The CBaseLeaf::GetBoundingBox method accepts two 3D vector references and populates them with the minimum and maximum extents of the leaf's AABB.

```
void CBaseLeaf::GetBoundingBox( D3DXVECTOR3 & Min, D3DXVECTOR3 & Max ) const
{
    // Retrieve the bounding boxes
    Min = m_vecBoundsMin;
    Max = m_vecBoundsMax;
}
```

The application will never need to set the bounding box of a leaf since that is the responsibility of the tree building process. Therefore, the SetBoundingBox method is not a member of ILeaf (the API used by the application); it is added to CBaseLeaf instead. The derived tree classes will call this method when the leaf is created at a terminal node and pass it the minimum and maximum extents of an AABB which represents the area of the terminal node (the leaf). Below we see the code to the function that will be called by the tree building process after a leaf has been allocated at a terminal node.

```
void CBaseLeaf::SetBoundingBox( const D3DXVECTOR3 & Min, const D3DXVECTOR3 & Max )
{
    // Store the bounding boxes
    m_vecBoundsMin = Min;
    m_vecBoundsMax = Max;
}
```

We have now covered all of the housekeeping methods in our CBaseLeaf class (the class that will be used to represent polygon and detail area data at terminal nodes in the tree). This object will be used by all of our derived tree classes to represent leaf data.

There are some additional members and methods that are not shown here as they relate to the rendering subsystem of CBaseTree. These will be discussed in the following lesson. For now, we have covered all the important code that will be needed to understand the construction and configuration of leaf data both during the build process and during collision queries.

14.11.5 CBaseTree – The Source Code

CBaseTree is the class that our other tree classes will be derived from. It implements many of the methods required by the ISpatialTree interface and as such, by deriving our tree classes from this class,

we avoid having to duplicate such housekeeping functionality. As discussed earlier, with CBaseTree in place, we can easily create almost any tree type we want simply by deriving a new class from it and implementing its build and query traversal functions. CBaseTree will be responsible for managing the polygon, leaf, and detail area lists and implementing the methods from ISpatialTree that allow the application to register data with the tree prior to the build process.

CBaseTree will also implement other methods that our derived classes will not want to worry about, such as the repair method from ISpatialTree. The CBaseTree::Repair method will repair T-junctions in the geometry and can be called after the tree has been built. Although we will not discuss the rendering logic in this chapter, this class also implements the DrawSubset method responsible for rendering the various subsets of static polygons stored within the tree. It also has an implementation of a ProcessVisibility method, although this method must be overridden in your derived class. The derived version must call the CBaseTree version prior to performing its visibility traversal. Although the meaning of the CBaseTree version of this method will not be fully understood until the next lesson, this method essentially gives CBaseTree a chance to flush its render buffers before they get refilled by the derived class. This will be explained later, so do not worry too much about it for now.

Finally, CBaseTree implements some utility methods that can be called by the derived class to make life easier. An example of this happens when implementing the DebugDraw method in your derived classes. As discussed earlier, the DebugDraw method traverses the hierarchy and renders a bounding box for any visible leaf node. This allows us to see each leaf's volume when running our application. However, although this function essentially has the same task to perform for each tree type, the method must be implemented for each tree type due to the fact that we traverse an oct-tree differently from how we traverse a quad-tree. But the traversal code is very small; it is really the rendering of the bounding box which takes a bit more code and will be identical for each tree type.

Because of this fact, CBaseTree will expose a method called DrawBoundingBox which can be called from a derived class and passed an AABB. This method will take case of the actual construction and rendering of the bounding box to the frame buffer. Therefore, all we have to do when we implement the DebugDraw method in our derived tree classes is write code in there that traverses the tree looking for visible leaf nodes. As we find each one, we just call the CBaseTree::DrawBoundingBox method and pass it the AABB of the leaf in question. This function will then render the bounding box and we minimize redundant code. This class also implements another DebugDraw helper function called CBaseTree::ScreenTint that can be passed a color and will alpha blend a quad over the entire frame buffer. We use this in our derived class's DebugDraw method to tint the screen red when the camera enters a leaf that has data contained in it (a non-empty leaf).

Below we see the class declaration of CBaseTree. We have removed any functions, structures, and member variables related to its rendering subsystem since we will introduce these in the following lesson.

```
class CBaseTree : public ISpatialTree
{
public:

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

```
    // Constructors & Destructors for This Class.
    virtual ~CBaseTree( );it easily
    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 void            DrawSubset         ( unsigned long nAttribID );
    virtual void            ProcessVisibility  ( CCamera & Camera );

protected:
    bool          PostBuild         ( );
     void         DrawBoundingBox   ( const D3DXVECTOR3 & Min,
                                      const D3DXVECTOR3 & Max,
                                      ULONG Color, bool bZEnable = false );

    void          DrawScreenTint    ( ULONG Color );
    void          CalculatePolyBounds ( );
    void          RepairTJunctions   ( CPolygon * pPoly1,
                                       CPolygon * pPoly2 );

    // Protected Variables for This Class.
    LPDIRECT3DDEVICE9          m_pD3DDevice;
    bool                       m_bHardwareTnL;
    LeafList                   m_Leaves;
    PolygonList                m_Polygons;
    DetailAreaList             m_DetailAreas;
};
```

A few things in the above declaration are worthy of node. First, notice how the CBaseLeaf::SetVisible method is made a friend, so that we can access it from within this class. The tree will need to be able to set the visible status of a leaf when it is performing its visibility pass. Also notice that it has a method called PostBuild. This method will be called by the derived class after the tree has been constructed. For example, our CQuadTree class will call this method at the very end of its Build function after the tree has been constructed. The PostBuild method does two things. The first thing it does is call the CBaseTree::CalculatePolyBounds method which will loop through each CPolygon stored in the tree, calculate its bounding box, and store that bounding box in the polygon structure. The polygon bounding boxes will be used by our broad phase collision step so that only polygons whose bounding boxes intersect the AABB of the swept sphere get passed onto the narrow phase. This is another example of a function that would otherwise need to be implemented in each of the derived classes if it were not for CBaseTree managing such functionality. The second thing the CBaseTree::PostBuild method does is call CBaseTree::BuildRenderData. This method is not shown above because we will discuss it in the next chapter (basically, it configures the rendering subsystem for CBaseTree).

We can see in the above declaration that the CBaseTree also has a method called RepairTJunctions. It is called from the CBaseTree::Repair method to mend any T-junctions which were introduced during the building phase. If the tree is not being used for rendering and you wish to repair T-junctions introduced in the build phase, then the application should call the Repair method after the tree has been built. If the tree is being used for rendering then there is no need, because the BuildRenderData method will

automatically call this method before preparing the render data. Keep in mind that if a tree is being used for rendering, we definitely want to always repair T-junctions to remove unsightly artifacts. We will examine what T-junctions are and how they can be repaired later in this lesson.

Let us now discuss the member variables declared in CBaseTree which will be used for storage by the derived classes.

LPDIRECT3DDEVICE9 m_pD3DDevice
If the application wishes to use the tree for rendering, it should pass a valid pointer to a Direct3D device into the constructor of the derived class, which will pass it along to the base class for storage in this member. If NULL is passed to the derived class constructor then NULL will be passed to the constructor of CBaseTree as well and this parameter will be set to NULL. If this parameter is NULL, no render data will be built when PostBuild is called at the end of the derived class's Build method.

bool m_bHardwareTnL
This boolean will also be set via the application passing its value to the derived class constructor, which in turn will be passed to the CBaseTree constructor and stored in this member. We have used a boolean value like this many times before to communicate to a component whether the device being used is a hardware or software vertex processing device. We will see in the following chapter how the base tree class will need to know this information when building the vertex and index buffers for its render data.

LeafList m_Leaves
PolygonList m_Polygons
DetailAreaList m_DetailAreas
Earlier we saw that the types LeafList, PolygonList, and DetailAreaList were defined in ISpatialTree as STL lists (linked lists) of ILeaf, CPolygon, and TreeDetailArea objects, respectively. These linked lists store the leaves, polygons, and detail areas when they are added to the tree. At any point (even pre-build) the m_Polygons and m_DetailAreas lists will contain all the polygons and details areas registered and in use by the tree at that time. m_Leaves will only contain valid data after the tree has been built and the leaves have been created.

Before the tree is built, the application will need a way to register detail areas and polygons with the tree so that they can be used in the building process. For example, every time we load a polygon from an IWF file we will call the AddPolygon method that will add it to the above list. After all the data has been added to the tree, but prior to the Build function being called, this list will contain all the polygon data that will be partitioned by the build process. After the build process has completed however, the polygons stored in the m_Polygons array (and in the leaf polygon arrays) may be different from the original list that existed prior to tree compilation. This is certainly true if a clipped tree is being constructed since many of the polygons in the original list will be deleted and replaced by two split fragments. Likewise, even if the lists are the same post-build, if you call the Repair function to mend T-junctions, additional triangles will be inserted to repair those T-junctions. The important point is that whether post-build or pre-build, the m_Polygons and m_DetailAreas lists will always contain all the polygons and detail areas being used by the tree. In the post-build case, we will also have pointers to all the polygons and detail areas in these lists stored in the polygon and detail area vectors of the leaves as well.

Constructor – CBaseTree

The CBaseTree constructor is called from the constructor of the derived class and is passed two parameters supplied by the application. If pDevice is not NULL then it means that the application wishes to use this tree for rendering and the rendering subsystem will be invoked (described in the next chapter). The device pointer and the boolean describing the hardware/software status of the device are stored in the base class variables and the reference count of the device is incremented.

```
CBaseTree::CBaseTree( LPDIRECT3DDEVICE9 pDevice, bool bHardwareTnL )
{
    // Store the D3D Device and other details
    m_pD3DDevice    = pDevice;
    m_bHardwareTnL  = bHardwareTnL;

    // Add ref the device (if available)
    if ( pDevice ) pDevice->AddRef();
}
```

There are a few additional lines to this constructor not shown here since they concern the rendering system which we will introduce later.

AddPolygon - CBaseTree

All of our derived classes will have a few things in common. One is the need for the application to add polygon data to its internal arrays in preparation for the build process. The AddPolygon method will be called by the application every time it loads a static polygon which it would like to be part of the data set that is to be spatially partitioned. If you look in the CScene::ProcessVertices function of Lab Project 14.1, you will see that a new line has been inserted that adds the vertices of the polygon currently being processed to the spatial tree being used by the scene. This is the function it calls, and as you can see, it simply adds the passed polygon pointer to its internal polygon list.

```
bool CBaseTree::AddPolygon( CPolygon * pPolygon )
{
    try
    {
        // Add to the polygon list
        m_Polygons.push_back( pPolygon );

    } // End Try Block
    catch ( ... )
    {
        return false;

    } // End Catch Block

    // Success!
    return true;
}
```

Only after the application has called the AddPolygon method for every static polygon it wishes to be stored in the tree should it call the tree's Build method. We will see later how the derived class's Build function will use the polygon list to build its spatial hierarchy.

AddDetailArea - CBaseTree

The application will also need the ability to register detail areas with the tree prior to the Build method being executed. This method is passed a TreeDetailArea structure which will be added to the tree's detail area list.

```
bool CBaseTree::AddDetailArea( const TreeDetailArea & DetailArea )
{
    try
    {
        // Allocate a new detail area structure
        TreeDetailArea * pDetailArea = new TreeDetailArea;
        if ( !pDetailArea ) throw std::bad_alloc();

        // Copy over the data from that specified
        *pDetailArea = DetailArea;

        // Add to the area list
        m_DetailAreas.push_back( pDetailArea );

    } // End Try Block

    catch ( ... )
    {
        return false;

    } // End Catch Block

    // Success!
    return true;
}
```

This method should be called by the application for each detail area (AABB) it would like to register with the tree prior to tree compilation. These detail areas can be used to force the tree to partition space that contains no static polygon data or not partition space that contains much polygon data. We will understand exactly how this is achieved later when we look at the Build methods for the various derived classes.

Retrieving Data from the Tree - CBaseTree

There may be times after the build process has been completed (or pre-build with respect to the polygon and detail area lists) when an application would like to retrieve a list of all the polygons, leaves, and

detail areas being used by the entire tree. The following three methods are simple access functions which return the leaf list, polygon list, and the detail area list back to the caller.

```
CBaseTree::LeafList & CBaseTree::GetLeafList()
{
    return m_Leaves;
}
```

```
CBaseTree::PolygonList & CBaseTree::GetPolygonList()
{
    return m_Polygons;
}
```

```
CBaseTree::DetailAreaList & CBaseTree::GetDetailAreaList()
{
    return m_DetailAreas;
}
```

AddLeaf - CBaseTree

Our derived classes will need a way to add leaves to their arrays during the build process. When a terminal node is encountered and a new CBaseLeaf is allocated and attached to the terminal node, we will also want to store that leaf's pointer in the tree's leaf list. Since this functionality will be the same for each tree type we will implement, this method in CBaseTree will save us the trouble of reinventing the wheel in each of our derived classes. The code simply adds the passed leaf pointer to the tree's internal leaf list.

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

    } // End Try Block

    catch ( ... )
    {
        return false;

    } // End Catch Block

    // Success!
    return true;
}
```

PostBuild - CBaseTree

The PostBuild method of CBaseTree should be called by the derived class after the tree has been constructed. In our derived classes, we will call this function at the very bottom of their Build method. PostBuild first calls the CBaseTree::CalculatePolyBounds method (which we will discuss next) which loops through each polygon in the tree's polygon list and calculates and stores its AABB. It then calls the CBaseTree::BuildRenderData method which allows the CBaseTree to initialize the render system with the tree data that has just been constructed. Many of you used to working with MFC might recognize this type of relationship, as it is much like creating a window. In MFC, methods can be overridden that allow your application to perform some default processing either just before or just after the window has been created. In this case, we are letting the base tree know that the tree building process is complete so that it can be prepared for rendering. The code to the function is shown below.

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

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

The CalculatePolyBounds method will be discussed next but we will defer our discussion of the BuildRenderData method until the next lesson where we discuss the rendering system. If the application did not pass a 3D device pointer into the derived tree's class constructor, this function will simply return without doing anything. In this case, the tree's rendering functionality will not be available but it can still be queried by the collision system.

CalculatePolyBounds - CBaseTree

The CalculatePolyBounds function will be used by all of our derived tree types. It will particularly important for an efficient broad phase collision step. Rather than ask the tree to return a list of leaves that intersect the swept sphere's AABB and the send the polygons in those leaves immediately on to the narrow phase, we will introduce an intermediate broad phase step which will be cheap but very effective. After we have retrieved the leaves that the swept sphere intersects, we will test each polygon in those leaves against the swept sphere AABB using their respective bounding boxes. Only polygons whose bounding boxes intersect the bounding box of the swept sphere will need to be passed to the more expensive narrow phase. This is an inexpensive test that gives us a very impressive performance enhancement.

> **Note:** Performing the polygon bounding box test in our lab testing really did increase speed by an impressive amount. For example, on one of the levels we were testing with the collision system, even when using the tree to collect only the relevant leaves, but without performing the polygon bounding box tests, our frame rates fell to ~30fps when querying a particularly dense leaf (i.e., a leaf with many polygons in it). After adding the polygon bounding box test, many polygons were rejected and did not get sent to the narrow phase and our frame rate in that same leaf increased to a solid 560fps. That is

obviously quite a savings and well worth the small amount of additional memory the bounding boxes add to each polygon.

This method is called from the CBaseTree::PostBuild method, which itself is called from the Build function of the derived class after the tree has been fully constructed. The job of the function is straightforward enough; loop through each static polygon stored in the tree (the m_Polygons STL linked list) and compile a bounding box for each one.

The firsts section of the function (shown below) creates an STL list iterator (a PolygonList iterator) which it uses to step through the elements in the polygon list. For each polygon, it aliases its bounding box minimum and maximum extents vectors with local variables Min and Max (for ease of use) and then sets the bounding box of the polygon to initially bogus values.

```
void CBaseTree::CalculatePolyBounds( )
{
    ULONG                    i;
    CPolygon                *pCurrentPoly;
    PolygonList::iterator    Iterator;

    // Calculate polygon bounds
    for ( Iterator = m_Polygons.begin(); Iterator != m_Polygons.end(); ++Iterator )
    {
        // Get poly pointer and bounds references for easy access
        pCurrentPoly = *Iterator;
        if ( !pCurrentPoly ) continue;
        D3DXVECTOR3 & Min = pCurrentPoly->m_vecBoundsMin;
        D3DXVECTOR3 & Max = pCurrentPoly->m_vecBoundsMax;

        // Reset bounding box
        Min = D3DXVECTOR3( FLT_MAX, FLT_MAX, FLT_MAX );
        Max = D3DXVECTOR3( -FLT_MAX, -FLT_MAX, -FLT_MAX );
```

Since FLT_MAX contains the maximum number that we can store in a float, and we set the maximum extents vector initially to the minimum possible value and the minimum extents vector to the maximum possible values, we create an initial huge “inside-out” box.

In the next section of the loop we test the position of each vertex in the polygon to see if it is contained within the box we have currently compiled. If not, the box will be adjusted to contain the vertex (this will always be the case for the first vertex due to the initial starting values of the box).

```
        // Build polygon bounds
        for ( i = 0; i < pCurrentPoly->m_nVertexCount; ++i )
        {
            CVertex & vtx = pCurrentPoly->m_pVertex[i];
            if ( vtx.x < Min.x ) Min.x = vtx.x;
            if ( vtx.y < Min.y ) Min.y = vtx.y;
            if ( vtx.z < Min.z ) Min.z = vtx.z;
            if ( vtx.x > Max.x ) Max.x = vtx.x;
            if ( vtx.y > Max.y ) Max.y = vtx.y;
            if ( vtx.z > Max.z ) Max.z = vtx.z;

        } // Next Vertex
```

Finally, before moving on to process the next polygon in the list, we will grow the box by 0.1 units in each direction to create a small buffer around the polygon. The reason we do this is to provide us a safety buffer when performing floating point tests. If two boxes (the polygon box and swept sphere box) were next to each other such that they were exactly touching, we risk the polygon being rejected for the narrow phase due to floating point accumulation/rounding errors. We certainly want to make sure that we do not reject a polygon from the narrow phase that we may be colliding with or we might be allowed to pass straight through it. By growing the box just slightly, we make this a fuzzy test where their boxes would overlap and the polygon would definitely be included in the case just mentioned.

```
        // Increase bounds slightly to relieve this operation during
        // intersection testing
        Min.x -= 0.1f; Min.y -= 0.1f; Min.z -= 0.1f;
        Max.x += 0.1f; Max.y += 0.1f; Max.z += 0.1f;

    } // Next Bounds
}
```

When this function, returns every polygon in the tree will contain a world space bounding box that the collision system (or any external component) can use to further refine the number of polygons in a leaf that get passed to the narrow phase.

14.11.6 Utility Methods - CBaseTree

These last two methods are not used by the application or by any other function in CBaseTree. They exist to make your life easier should you choose to implement the DebugDraw method in your derived classes. It is purely optional that you implement this method since ISpatialTree provides a default implementation that does nothing. Therefore, the application can always call this method even if you have not implemented it and no harm will be done.

The DebugDraw method will be implemented in all our derived classes in a very similar way. As discussed earlier, it will traverse the tree searching for visible leaf nodes. When a leaf node is found, we will draw a bounding box around the area represented by the leaf. We will also tint the screen red if the camera is currently contained in a leaf that contains polygon data. Since most of the code is geared towards generating and rendering the bounding box and handling the screen tinting, we decided to add the two functions that perform these tasks to CBaseTree (DrawBoundingBox and DrawScreenTint) and avoid duplicating code unnecessarily.

DrawBoundingBox – CBaseTree

The CBaseTree::DrawBoundingBox method will be called by the DebugDraw method in each of our derived classes (assuming the scene is calling ISpatialTree::DebugDraw). It will be passed the world space bounding box (the minimum and maximum extent vectors) of the leaf and (as its third parameter) the color we would like to render the wireframe box. The fourth parameter is a boolean that will indicate whether we would like the box to be rendered using the depth buffer.

When our DebugDraw routines call this method for a visible leaf which the camera is not currently in, the box will be rendered normally with depth testing enabled. For the current camera leaf we will pass false as this parameter and render it to the screen without depth testing. This ensures that it will always be rendered on top of anything already in the frame buffer, making it easier to see the box for the leaf we are currently standing in without it being obscured by nearby geometry.

The function has two static members. The first is an array of 24 vertices that will define the four vertices of each face of the cube (6 faces * 4 vertices = 24). We will also use a static boolean called BoxBuilt to indicate whether we have already built the box in the previous call. Since these are static methods they will retain their values each time the function is called. We do this so that the box mesh is only ever built the first time the method is ever called. If this method has never been called before, then the static BoxBuilt will be set to false and code will be executed to generate the box vertices and add them to the static vertex array. The BoxBuilt boolean will then be set to true so that the next time the method is called it will know that the box mesh has already been constructed and we will not have to do it again. Since the box we wish to draw will be a wireframe box, the vertices in this array will describe a list of line primitives which will be rendered using the D3DPT_LINELIST primitive type. The line list mesh will be stored in the vertices array as a 1x1x1 cube and it will be transformed and scaled to the size and position of the current leaf node. Let us look at this function in a few sections.

In the first section of the function we can see that a local structure is defined to describe what a box vertex looks like. Basically, it contains a position and a color. We can also see that if the tree does not currently store a pointer to a 3D device, then this is not a tree that is intended to be rendered and we return.

```
void CBaseTree::DrawBoundingBox( const D3DXVECTOR3 & Min,
                                 const D3DXVECTOR3 & Max,
                                 ULONG Color,
                                 bool bZEnable /* = false */ )
{
    struct BoxVert
    {
        D3DXVECTOR3 Pos;
        ULONG       Color;
    };

    ULONG          i;
    static BoxVert Vertices[24];
    static bool    BoxBuilt = false;
    D3DXMATRIX     mtxBounds, mtxIdentity;
    D3DXVECTOR3    BoundsCenter, BoundsExtents;
    ULONG          OldStates[6];

    // If there is no device, we can't draw
    if ( !m_pD3DDevice ) return;
```

Next we create an identity matrix (which will be used in a moment) and if the BoxBuilt boolean is false, we fill in the elements of the static vertex array. Notice that we position the 24 vertices such that they describe the vertices of each face in a unit sized cube; a cube of 1 unit in size in each direction which is centered at (0, 0, 0). This means the cube vertices are in the -0.5 to +0.5 range along each axis. We can think of this cube at this point as being in model space.

```
    // We need an identity matrix later
    D3DXMatrixIdentity( &mtxIdentity );

    // Build the box vertices if we have not done so already
    if ( !BoxBuilt )
    {
        // Bottom 4 edges
        Vertices[0].Pos  = D3DXVECTOR3( -0.5f, -0.5f, -0.5f );
        Vertices[1].Pos  = D3DXVECTOR3(  0.5f, -0.5f, -0.5f );

        Vertices[2].Pos  = D3DXVECTOR3(  0.5f, -0.5f, -0.5f );
        Vertices[3].Pos  = D3DXVECTOR3(  0.5f, -0.5f,  0.5f );

        Vertices[4].Pos  = D3DXVECTOR3(  0.5f, -0.5f,  0.5f );
        Vertices[5].Pos  = D3DXVECTOR3( -0.5f, -0.5f,  0.5f );

        Vertices[6].Pos  = D3DXVECTOR3( -0.5f, -0.5f,  0.5f );
        Vertices[7].Pos  = D3DXVECTOR3( -0.5f, -0.5f, -0.5f );

        // Top 4 edges
        Vertices[8].Pos  = D3DXVECTOR3( -0.5f,  0.5f, -0.5f );
        Vertices[9].Pos  = D3DXVECTOR3(  0.5f,  0.5f, -0.5f );

        Vertices[10].Pos = D3DXVECTOR3(  0.5f,  0.5f, -0.5f );
        Vertices[11].Pos = D3DXVECTOR3(  0.5f,  0.5f,  0.5f );

        Vertices[12].Pos = D3DXVECTOR3(  0.5f,  0.5f,  0.5f );
        Vertices[13].Pos = D3DXVECTOR3( -0.5f,  0.5f,  0.5f );

        Vertices[14].Pos = D3DXVECTOR3( -0.5f,  0.5f,  0.5f );
        Vertices[15].Pos = D3DXVECTOR3( -0.5f,  0.5f, -0.5f );

        // 4 Side 'Struts'
        Vertices[16].Pos = D3DXVECTOR3( -0.5f, -0.5f, -0.5f );
        Vertices[17].Pos = D3DXVECTOR3( -0.5f,  0.5f, -0.5f );

        Vertices[18].Pos = D3DXVECTOR3(  0.5f, -0.5f, -0.5f );
        Vertices[19].Pos = D3DXVECTOR3(  0.5f,  0.5f, -0.5f );

        Vertices[20].Pos = D3DXVECTOR3(  0.5f, -0.5f,  0.5f );
        Vertices[21].Pos = D3DXVECTOR3(  0.5f,  0.5f,  0.5f );

        Vertices[22].Pos = D3DXVECTOR3( -0.5f, -0.5f,  0.5f );
        Vertices[23].Pos = D3DXVECTOR3( -0.5f,  0.5f,  0.5f );

        // We're done
        BoxBuilt = true;

    } // End if box has not yet been built
```

At this point all 24 model space box vertices have been placed in the vertex array and the BoxBuilt boolean will be set to true. Thus, the above code will not be performed the next time this function is called to render another leaf box.

In the next section we loop through each of the 24 vertices and set their color value to the color parameter passed by the caller (the DebugDraw method of the derived class).

```
    // Set the color to that specified.
    for ( i = 0; i < 24; ++i ) Vertices[i].Color = Color;
```

In the next section we need to create a matrix that, when set on the device as a world matrix, will transform and scale the model space vertices such that the box is transformed into a box that is equal in size to the passed box vectors (the leaf's bounding box) and positioned in the world such that its center point is at the center point of the leaf.

First we need to find the world space center point of the leaf bounding box, which we can calculate by adding the leaf's world space minimum and maximum box vector and dividing the result by 2. We then calculate the size of the box (the diagonal size of the leaf's bounding box) by subtracting the minimum extent vector from the maximum vector.

```
    // Compute the bound's centre point and extents
    BoundsCenter  = (Min + Max) / 2.0f;
    BoundsExtents = Max - Min;
```

At this point BoundsExtents contains the diagonal length of the leaf bounding box. Since our box mesh is defined to be a 1x1x1 unit square (0.5 units in each dimension), if we store the BoundsExtents in the diagonal of a matrix, we will have a scaling matrix that will transform the vertices of the box mesh so that it becomes the same size as the leaf box in world space.

For example, let us imagine that the world space leaf bounding box extents are the two vectors (50, 50, 50) and (60, 60, 60). In this case, BoundsExtents will be:

$$(60, 60, 60) - (50, 50, 50) = (10, 10, 10)$$

If we store the x, y, and z components of the resulting vector in the diagonal of a matrix, we will get a matrix which will scale any x, y, and z vertex components by 10. Since the box is defined in the -0.5 to 0.5 range along each axis, we can see that when the box vertices are multiplied by this scaling matrix, it will result in vertices in the -0.5*10=-5 to 0.5*10=5 range (i.e., a box in the range of [-5, 5] along each axis). This is exactly as it should be as it matches the size of the leaf bounding box. All we have to do now is place the world space position vector of the center of the leaf (BoundsCenter) in the translation row of the matrix and we will have a world matrix that will properly scale and position the model space box mesh to match the leaf.

```
    // Build the scaling matrix
    D3DXMatrixScaling( &mtxBounds,
                       BoundsExtents.x,
                       BoundsExtents.y,
                       BoundsExtents.z );

    // Translate the bounding box matrix
    mtxBounds._41 = BoundsCenter.x;
    mtxBounds._42 = BoundsCenter.y;
    mtxBounds._43 = BoundsCenter.z;
```

```
    // Set the bounding box matrix
    m_pD3DDevice->SetTransform( D3DTS_WORLD, &mtxBounds );
```

At this point we have the world matrix set on the device and we are almost ready to render, but first we will need to set some render states. We will start by disabling Z buffer writing (we do not want our box lines to obscure real geometry) and enable or disable depth testing based on the boolean parameter passed in. We will also want to disable lighting (our lines are constructed from pre-colored vertices) and set the first texture/color stage so that only the diffuse color of the vertex is used. Of course, we had better retrieve and backup the current state settings because we would not want to change something that will cause the rest of the application to render incorrectly. So we will retrieve the current states that we intend to change and store them in a local states array (OldStates) as shown below.

```
    // Retrieve old states
    m_pD3DDevice->GetRenderState( D3DRS_ZWRITEENABLE, &OldStates[0] );
    m_pD3DDevice->GetRenderState( D3DRS_ZENABLE, &OldStates[1] );
    m_pD3DDevice->GetRenderState( D3DRS_LIGHTING, &OldStates[2] );
    m_pD3DDevice->GetRenderState( D3DRS_COLORVERTEX, &OldStates[3] );
    m_pD3DDevice->GetTextureStageState( 0, D3DTSS_COLORARG1, &OldStates[4] );
    m_pD3DDevice->GetTextureStageState( 0, D3DTSS_COLOROP, &OldStates[5] );
```

With the current states currently backed up we will setup the render states we wish to use to render our box edges.

```
    // Setup new states
    m_pD3DDevice->SetRenderState( D3DRS_ZWRITEENABLE, FALSE );
    m_pD3DDevice->SetRenderState( D3DRS_ZENABLE, bZEnable );
    m_pD3DDevice->SetRenderState( D3DRS_LIGHTING, FALSE );
    m_pD3DDevice->SetRenderState( D3DRS_COLORVERTEX, TRUE );
    m_pD3DDevice->SetTextureStageState( 0, D3DTSS_COLORARG1, D3DTA_DIFFUSE );
    m_pD3DDevice->SetTextureStageState( 0, D3DTSS_COLOROP, D3DTOP_SELECTARG1 );
```

With the pipeline now configured, we set the FVF of our box vertices and render. Since our box vertices contain positional information and a diffuse color we will inform the pipeline by using the D3DFVF_XYZ | D3DFVF_DIFFUSE flag combination. We will then render the vertices straight from the box vertex array using the DrawPrimitiveUP function. UP stands for ‘User Pointer’ and it allows us to render straight from system memory without having to store the vertices in vertex buffers. This is a highly inefficient way to render primitives, but it is acceptable in this case since it is only a debug routine and it makes the implementation a lot easier.

```
    // Draw
    m_pD3DDevice->SetFVF( D3DFVF_XYZ | D3DFVF_DIFFUSE );
    m_pD3DDevice->DrawPrimitiveUP( D3DPT_LINELIST, 12, &Vertices, sizeof(BoxVert));
```

As you can see in the above code, we ask DrawPrimitiveUP to draw 12 lines, where each line consists of two vertices (start and end points). Thus the vertex array we pass in as the third parameter must contain at least 12*2=24 vertices, which we know ours does. The fourth parameter is the stride of the vertex structure we are using so the pipeline knows how many bytes to advance when stepping from vertex to vertex during the transformation process.

With all lines rendered for the currently passed leaf node, we restore the render states we backed up earlier and reset the world matrix to identity.

```
    // Reset old states
    m_pD3DDevice->SetRenderState( D3DRS_ZWRITEENABLE, OldStates[0] );
    m_pD3DDevice->SetRenderState( D3DRS_ZENABLE, OldStates[1] );
    m_pD3DDevice->SetRenderState( D3DRS_LIGHTING, OldStates[2] );
    m_pD3DDevice->SetRenderState( D3DRS_COLORVERTEX, OldStates[3] );
    m_pD3DDevice->SetTextureStageState( 0, D3DTSS_COLORARG1, OldStates[4] );
    m_pD3DDevice->SetTextureStageState( 0, D3DTSS_COLOROP, OldStates[5] );

    // Reset the matrix
    m_pD3DDevice->SetTransform( D3DTS_WORLD, &mtxIdentity );
}
```

DrawScreenTint – CBaseTree

This is utility function tints the screen a red color when the camera is positioned inside a populated leaf node. It accepts one parameter: a DWORD containing the color to tint the screen.

This technique should be familiar to you since we did the same thing in Chapter 7 of Module I when we implemented our underwater effect. We just create four transformed and lit vertices (screen space vertices with diffuse color components) and position them at the four corners of the viewport. We then render the quad to the screen with alpha blending enabled (and with the depth buffer disabled so that it is definitely rendered on top of everything else) to blend the color of the quad over the entire scene stored in the frame buffer..

The first part of the function is shown below. Note that we define a vertex structure (TintVert) which is laid out as a transformed and lit vertex. We know that when we pass a pre-transformed (viewport space) vertex to the pipeline, its positional vector is 4D, not 3D. The first two positional components of the vertex (x and y) describe the position on the viewport where the vertex is positioned. The third component (z) contains the depth buffer value in the range of 0.0 to 1.0. Since we are going to disable depth testing and writing when we render this quad, we will just set this to zero. The fourth positional component is called RHW by DirectX and should contain the reciprocal of homogenous W (i.e., 1 divided by the viewspace vertex Z component). This is all designed to work if you are performing your own transformation of geometry and merely wish DirectX to render the screen space polygons. 1/W gets closer to 1 the closer to the near plane the vertex is. It describes the depth of the vertex with respect to the camera. Recall that this value is used during rendering by the DirectX fog engine. However, we just wish to draw a quad on the screen without fog, so we will just set this to 1 which simulates a vertex very close to the near plane. However, since we are not using fog and the depth buffer is disabled, we could actually this value to anything without any ill effect.

```
void CBaseTree::DrawScreenTint( ULONG Color )
{
    struct TintVert
    {
        D3DXVECTOR4 Pos;
        ULONG       Color;
```

```
    };

    ULONG        i;
    TintVert     Vertices[4];
    ULONG        OldStates[10];
    D3DVIEWPORT9 Viewport;


    // If there is no device, we can't draw
    if ( !m_pD3DDevice ) return;

    // Retrieve the viewport dimensions
    m_pD3DDevice->GetViewport( &Viewport );


    // 4 Screen corners
    Vertices[0].Pos=D3DXVECTOR4((float)Viewport.X, (float)Viewport.Y, 0.0, 1.0f );

    Vertices[1].Pos=D3DXVECTOR4((float)Viewport.X + Viewport.Width,
                                (float)Viewport.Y, 0.0, 1.0f );

    Vertices[2].Pos  = D3DXVECTOR4( (float)Viewport.X + Viewport.Width,
                                    (float)Viewport.Y + Viewport.Height,
                                    0.0, 1.0f );

    Vertices[3].Pos  = D3DXVECTOR4( (float)Viewport.X,
                                    (float)Viewport.Y + Viewport.Height,
                                    0.0, 1.0f );

    // Set the color to that specified.
    for ( i = 0; i < 4; ++i ) Vertices[i].Color = Color;
```

The above code shows how we fetch the device viewport and retrieve the viewport rectangle. We will use this rectangle to position the vertices of our quad in the frame buffer. As you can see, the four vertices are positioned at the top left, top right, bottom right, and bottom left of the viewport. The z component of each vertex is set to 0.0 and the RHW component is set to 1.0. We then loop through each of the vertices and set its color properties to the color value passed into the function.

Next we will backup and set the render states and texture states we wish to use to render our screen effect. We will disable Z writing and testing, disable lighting, and enable alpha blending. We will set the source and destination color blending results to use the alpha component of the passed color to weight the blending between the color of the quad and the color already in the frame buffer. This means we need to set the source and destination blend render states to D3DBLEND_SRCALPHA and D3DBLEND_INVSRCALPHA and configure texture stage zero to take its alpha and color components from the diffuse vertex color. This will generate a final color for each pixel at the end of the texture stage cascade which has color and alpha components equal to the diffuse and alpha components of the passed color, respectively. We then pass this to the renderer where the alpha component will be used as the weighting factor with the blending modes we have configured.

Below we see the code that makes a backup of all the states we intend to change, sets all the new states, renders the quad, and then restores the original device states.

```
    // Retrieve old states
    m_pD3DDevice->GetRenderState( D3DRS_ZWRITEENABLE, &OldStates[0] );
    m_pD3DDevice->GetRenderState( D3DRS_ZENABLE, &OldStates[1] );
    m_pD3DDevice->GetRenderState( D3DRS_LIGHTING, &OldStates[2] );
    m_pD3DDevice->GetRenderState( D3DRS_COLORVERTEX, &OldStates[3] );
    m_pD3DDevice->GetRenderState( D3DRS_ALPHABLENDENABLE, &OldStates[4] );
    m_pD3DDevice->GetRenderState( D3DRS_ALPHATESTENABLE, &OldStates[5] );
    m_pD3DDevice->GetTextureStageState( 0, D3DTSS_COLORARG1, &OldStates[6] );
    m_pD3DDevice->GetTextureStageState( 0, D3DTSS_COLOROP, &OldStates[7] );
    m_pD3DDevice->GetTextureStageState( 0, D3DTSS_ALPHAARG1, &OldStates[8] );
    m_pD3DDevice->GetTextureStageState( 0, D3DTSS_ALPHAOP, &OldStates[9] );

    // Setup new states
    m_pD3DDevice->SetRenderState( D3DRS_ZWRITEENABLE, FALSE );
    m_pD3DDevice->SetRenderState( D3DRS_ZENABLE, FALSE );
    m_pD3DDevice->SetRenderState( D3DRS_LIGHTING, FALSE );
    m_pD3DDevice->SetRenderState( D3DRS_COLORVERTEX, TRUE );
    m_pD3DDevice->SetRenderState( D3DRS_ALPHABLENDENABLE, TRUE );
    m_pD3DDevice->SetRenderState( D3DRS_ALPHATESTENABLE, FALSE );
    m_pD3DDevice->SetRenderState( D3DRS_SRCBLEND, D3DBLEND_SRCALPHA );
    m_pD3DDevice->SetRenderState( D3DRS_DESTBLEND, D3DBLEND_INVSRCALPHA );
    m_pD3DDevice->SetTextureStageState( 0, D3DTSS_COLORARG1, D3DTA_DIFFUSE );
    m_pD3DDevice->SetTextureStageState( 0, D3DTSS_COLOROP, D3DTOP_SELECTARG1 );
    m_pD3DDevice->SetTextureStageState( 0, D3DTSS_ALPHAARG1, D3DTA_DIFFUSE );
    m_pD3DDevice->SetTextureStageState( 0, D3DTSS_ALPHAOP, D3DTOP_SELECTARG1 );

    // Draw
    m_pD3DDevice->SetFVF( D3DFVF_XYZRHW | D3DFVF_DIFFUSE );
    m_pD3DDevice->DrawPrimitiveUP(  D3DPT_TRIANGLEFAN,
                                    2,
                                    &Vertices,
                                    sizeof(TintVert) );

    // Reset old states
    m_pD3DDevice->SetRenderState( D3DRS_ZWRITEENABLE, OldStates[0] );
    m_pD3DDevice->SetRenderState( D3DRS_ZENABLE, OldStates[1] );
    m_pD3DDevice->SetRenderState( D3DRS_LIGHTING, OldStates[2] );
    m_pD3DDevice->SetRenderState( D3DRS_COLORVERTEX, OldStates[3] );
    m_pD3DDevice->SetRenderState( D3DRS_ALPHABLENDENABLE, OldStates[4] );
    m_pD3DDevice->SetRenderState( D3DRS_ALPHATESTENABLE, OldStates[5] );
    m_pD3DDevice->SetTextureStageState( 0, D3DTSS_COLORARG1, OldStates[6] );
    m_pD3DDevice->SetTextureStageState( 0, D3DTSS_COLOROP, OldStates[7] );
    m pD3DDevice->SetTextureStageState( 0, D3DTSS ALPHAARG1, OldStates[8] );
    m_pD3DDevice->SetTextureStageState( 0, D3DTSS_ALPHAOP, OldStates[9] );
}
```

We have now covered all the house keeping and utility building code from CBaseTree that needs to be covered in this lesson. The one exception is the Repair method that performs T-junction repair, but we will come back to that a little later in this lesson and devote an entire section to it. In the next chapter we will add a rendering system to this class, but for now we have everything we need in place to create our derived tree classes and use them to perform efficient collision queries.

Before we discuss T-junction repair and move on to look at the implementations of each of the derived classes, we will need to extend our CCollision library with a few more intersection routines that will be

used by the tree building and query functions in the derived classes. For example, we will need to be able to determine when a point is inside an AABB, when two AABBs are intersecting, and when a ray intersects an AABB. We will discuss the implementations of these functions next, and we will make them static members of the CCollision class so that they can be used even if the collision system is not being used.

14.12 Point/AABB Intersection

One of the simplest collision tests we can perform is determining whether a point is inside an AABB. Such a method is used to find out if a certain position is contained inside a leaf node's bounds, as one obvious example. Our DebugDraw routines will use this intersection test to determine whether the leaf currently being visited contains the camera position. If it does, then it renders the bounding box of the leaf a different color.

The test is easy and cheap to perform and involves nothing more than separately testing the x, y, and z components of the passed position vector to see if they fall between the minimum and maximum x, y and z extents of the AABB. If any component of the position vector is outside the range of its corresponding components in the AABB extent vectors, the point is not contained and we return false. That is, the test is essentially stating "If the x component of the point is between the minimum x extent and the maximum x extent of the box, we have passed the X axis test and the two other axes must be tested, etc."

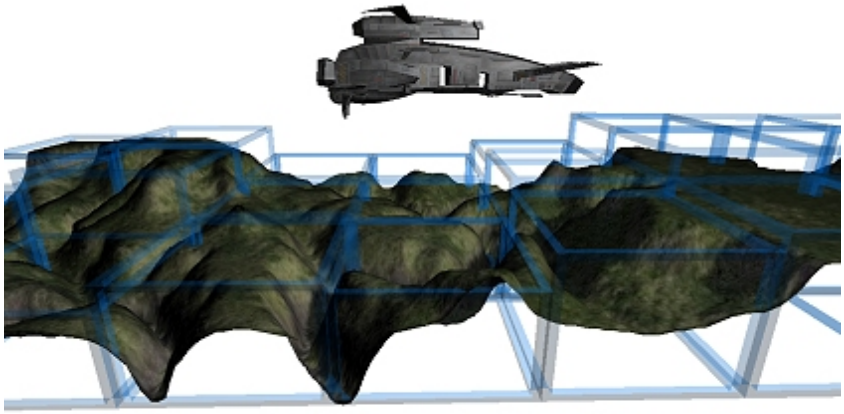
The code to PointInAABB is just a few lines long. The function only returns true if none of the three axis tests fail.

```
bool CCollision::PointInAABB( const D3DXVECTOR3& Point,
                              const D3DXVECTOR3& Min,
                              const D3DXVECTOR3& Max,
                              bool bIgnoreX /* = false */,
                              bool bIgnoreY /* = false */,
                              bool bIgnoreZ /* = false */ )
{
    // Does the point fall outside any of the AABB planes?
    if ( !bIgnoreX && (Point.x < Min.x || Point.x > Max.x) ) return false;
    if ( !bIgnoreY && (Point.y < Min.y || Point.y > Max.y) ) return false;
    if ( !bIgnoreZ && (Point.z < Min.z || Point.z > Max.z) ) return false;

    // We are behind all planes
    return true;
}
```

Notice how this method not only accepts the point and the two AABB extents as parameters, it also accepts three booleans allowing us to instruct the function to ignore tests along any axes. For example, if we pass in true for the bIgnoreY parameter, the Y component of the point will not be tested against the Y components of the AABB extents (i.e., the function will behave as if the Y axis test has not failed and that the point always falls within the box with respect to the Y axis).

It is quite useful to have the ability to ignore certain components in the test. Indeed, you will see that this is a feature we provide in all of our AABB intersection tests. It is particularly handy when performing a collision query on a quad-tree. As discussed earlier, the vanilla quad-tree will always have leaf nodes of the same height (the max vertical range of the entire level). However, we also discussed (and will implement) the Y-variant quad-tree, where the vertical extents of each node fit the contained geometry. This can produce certain problems when querying the tree since there may be times when a given position is contained inside no leaf node. This is actually true of any tree type but is more easily demonstrated with the Y-variant tree.

For example, Figure 14.53 shows a terrain compiled into a Y-variant quad-tree. We also see a space ship hovering above the terrain.



Multi Y Variant Quadtree

Figure 14.53

Imagine that our space ship has a spotlight mounted on the bottom of its hull, such that it illuminates the terrain immediately below it. In order to find the terrain polygons we need to illuminate, we would normally send the bounding box of the space ship down the tree and collect all the leaves it intersects. We can then fetch the polygons from those leaves and perform some custom lighting effect. However, in this case we can see that if we were to feed the bounding box of the ship into the tree, no leaves would be returned because it is not contained in any of the leaves. But, if we did the same test and ignored the Y component in each leaf (only testing the X and Z extents of the box), then the leaves beneath the ship would be returned even though the ship is not technically contained within their volume.

14.13 AABB/AABB Intersection

Another collision query that we will need to use quite a bit in our applications determines whether two AABBs intersect. This collision routine will be the heart of the CollectLeavesAABB method in our derived classes, which essentially traverses the tree with the passed query volumes and finds which leaves the passed bounding box intersects. At each leaf we will perform an AABB/AABB test and only add the leaf to the output list if there is an intersection and the collision function returns true.

Much like the PointInAABB test described above, the AABBIntersectAABB intersect function performs its tests on a per axis basis. If at any point we find an axis on which an overlap does not occur, the boxes do not overlap and the function can return false immediately. If there is overlap on each of the three axes, then the bounding boxes intersect one another and we return true. The test for an overlap on a given axis is very simple when dealing with AABBs as Figure 14.54 shows.

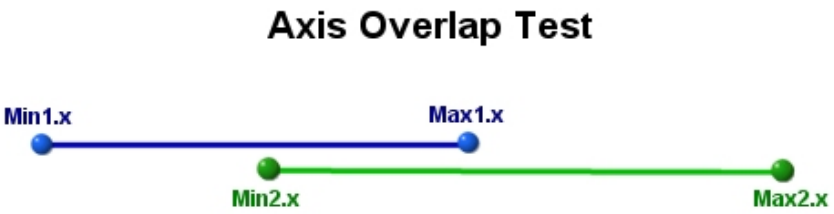


Figure 14.54

Figure 14.54 shows the X axis overlap test for two boxes that do indeed overlap. Since we are dealing with one axis at a time, the two boxes can be treated component-wise as lines on those axes. If Min1 and Max1 are the extent vectors of the first box and Min2 and Max2 are the extent vectors of the second box, we can see that if Min1.x is smaller than Max2.X and Max1.x is larger than Min2.x, then the boxes overlap on that axis. You should be able to imagine that if we were to move the blue line to the left so that they no longer overlapped, min2.x would no longer be smaller than Max1.x and the test would fail. That is, we would not have an overlap on the X axis which means the boxes cannot possibly be intersecting one another. When this is the case we can return false immediately without performing any other axis overlap tests.

The code is shown below. Note that it also supports the axis ignore functionality discussed in the previous function. If we choose to ignore any axis, then those axes are assumed to be overlapping.

```
bool CCollision::AABBIntersectAABB( const D3DXVECTOR3& Min1,
                                    const D3DXVECTOR3& Max1,
                                    const D3DXVECTOR3& Min2,
                                    const D3DXVECTOR3& Max2,
                                    bool bIgnoreX /* = false */,
                                    bool bIgnoreY /* = false */,
                                    bool bIgnoreZ /* = false */ )
{
    return (bIgnoreX || Min1.x <= Max2.x) && (bIgnoreY || Min1.y <= Max2.y) &&
           (bIgnoreZ || Min1.z <= Max2.z) && (bIgnoreX || Max1.x >= Min2.x) &&
           (bIgnoreY || Max1.y >= Min2.y) && (bIgnoreZ || Max1.z >= Min2.z);
}
```

The parameter list to the function includes the minimum and maximum extent vectors of the first AABB to test followed by the minimum and maximum extent vectors of the second AABB to test. Following this are three optional axis ignore booleans allowing us to ignore any of the axes during the test.

We have decided to also provide an overloaded version of this function that accepts a boolean reference as its first parameter that can be used to communicate back to the caller whether or not box 2 was completely contained inside box 1. This makes the test slower since it has to perform the containment test first, followed by the intersection test described above. However, there are times when dealing with

hierarchy traversal that this can prevent many future AABB/AABB intersection tests from needing to be performed.

For example, imagine that our application called the CollectLeavesAABB function for one of our derived tree classes. We know this function has the task of stepping through the tree and at each node testing for an intersection between the passed AABB and the AABB of the node/leaf. If the query AABB does not intersect a node's AABB then we do not have to bother traversing into any of its children, allowing us to reject portions of the tree from having to be testing and collected. However, if the query volume does intersect the node volume, we will want to step into its children. Whenever a leaf node is reached, we add its leaf structure to a list that can be passed back to the caller. However, with this new overloaded version of the function we can perform a optimization to our CollectLeavesAABB routine. If we know that the bounding box of a node is completely contained inside the query volume, then we already know that all of its child nodes (including the leaf nodes) will be contained in it also. Thus, we do not need to perform any further AABB intersection tests on any of its child nodes. Instead we can just traverse into its children searching for leaf nodes which, once found, are immediately added to the leaf list that will be returned.

The overloaded function is shown below with the containment tests performed at the beginning of the function. The passed bContained boolean is initially set to true, meaning that box 2 is fully contained in box 1 and each of the containment tests tries to find proof that box 2 pierces the bounds of box 1 and is therefore not fully contained. This is once again done of a per axis basis, so for example, when testing the X axis for containment, but we are essentially just trying to prove that either the minimum extents of box 2 are smaller then the minimum extents of box 1 or that the maximum extents of box 2 are larger than the maximum extents of box 1. If this is true for any axis, then box 2 is not contained in box1 and the containment boolean is set to false. When this is the case, we must perform the intersection test described above to test if the boxes even intersect. The intersection test at the end of the function only has to be performed if the containment test failed. As soon as we have proof that box 2 is contained in box 1 we can return true for intersection with the bContainment boolean set to true.

```
bool CCollision::AABBIntersectAABB( bool & bContained,
                                    const D3DXVECTOR3& Min1,
                                    const D3DXVECTOR3& Max1,
                                    const D3DXVECTOR3& Min2,
                                    const D3DXVECTOR3& Max2,
                                    bool bIgnoreX /* = false */,
                                    bool bIgnoreY /* = false */,
                                    bool bIgnoreZ /* = false */ )
{
    // Set to true by default
    bContained = true;

    // Is box contained totally inside
    if ( !bIgnoreX && (Min2.x < Min1.x || Min2.x > Max1.x) ) bContained = false;
    else
    if ( !bIgnoreY && (Min2.y < Min1.y || Min2.y > Max1.y) ) bContained = false;
    else
    if ( !bIgnoreZ && (Min2.z < Min1.z || Min2.z > Max1.z) ) bContained = false;
    else
    if ( !bIgnoreX && (Max2.x < Min1.x || Max2.x > Max1.x) ) bContained = false;
    else
    if ( !bIgnoreY && (Max2.y < Min1.y || Max2.y > Max1.y) ) bContained = false;
```

```
    else
    if ( !bIgnoreZ && (Max2.z < Min1.z || Max2.z > Max1.z) ) bContained = false;

    // Return immediately if it's fully contained
    if ( bContained == true ) return true;

    // Perform full intersection test
    return (bIgnoreX || Min1.x <= Max2.x) && (bIgnoreY || Min1.y <= Max2.y) &&
           (bIgnoreZ || Min1.z <= Max2.z) && (bIgnoreX || Max1.x >= Min2.x) &&
           (bIgnoreY || Max1.y >= Min2.y) && (bIgnoreZ || Max1.z >= Min2.z);
}
```

14.14 Ray/AABB Intersection

We have talked a lot about the usefulness of being able to send a ray through the tree and get back a list of intersecting leaves. One example of where such a technique will be used is in Module III when we will write a lightmap compiler. Beams of light emanating from a light source will be modelled as rays and we will need to determine as quickly as possible whether any polygons in the scene block that ray. As our scenes will likely involve of tens of thousands of polygons or more, we will be performing literally millions of these tests. To make sure that our lightmap compiler will compile its texture data as quickly as possible, the scene will first be compiled into a spatial tree so that we can query the scene efficiently by performing the ray query on the tree. There are plenty of other examples, but this should give you further indication that spatial partitioning will be an important tool in many areas.

When examining the ISpatialTree interface we saw that our derived tree classes will be expected to implement a method called CollectLeavesRay. This method will be passed a ray and will traverse the tree testing which leaves the ray intersects. These leaves will then be returned so that the calling application can access the polygon data. In the lightmap compiler discussed above, the leaves returned will contain the polygons that have the potential to block the ray. This handful of polygons can then be tested more closely using ray/polygon intersection tests.

The heart of the CollectLeavesRay function (at least for the derived classes we implement) will be the testing of a ray to see if it intersects the bounding box of a given node. If the ray does intersect a node then we have to traverse into its children and continue testing. If it does not, then we can abandon that branch of the tree and all the leaf data it contains. Whenever we enter a leaf node, the leaf object will be added to a list that will be returned to the caller. Thus, the function will return to the application a list of the leaves the ray intersected. It is clear then that we must learn how to test for an intersection between a ray and an axis aligned bounding box.

14.14.1 The Slabs Method

We will be using the slabs method for intersection testing since it will works for both axis aligned bounding boxes and oriented bounding boxes. Our implementation of this method however will be optimized to take advantage of the fact that we will only be using it for AABB testing. As such, our

function is to be used only for the testing of AABBs. Although we will be implementing a faster AABB only method, the intersection theory we discuss here should make it a fairly simple matter for you to implement a version of the function that works with OBBs as well.

The slabs method works by calculating the intersection between the ray and each set of parallel planes described by the cube faces. A slab is therefore a pair of faces that are parallel to one another. If we think of an axis aligned bounding box, it would be comprised of three slabs. The first slab might be the pair of planes described by the front and back face of the cube, whose normals are aligned to the coordinate system Z axis. The second slab might be the planes of the right and left faces whose normals are aligned with the coordinate system X axis. The third slab would be comprised of the planes of the top and bottom faces whose normals are aligned with the Y axis of the coordinate system.

The test can easily be converted to two dimensions just by dropping tests against the slab whose planes are aligned with the coordinate system Z axis. To make visualizing the slabs method a little easier, we will discuss the two dimensional case in our images. The only difference is that in the three dimensional case, we are testing three slabs instead of two.

Figure 14.55 shows the minimum and maximum extent vectors of an AABB which describe the bottom left and top right corners of a 2D axis aligned bounding box. It also shows the ray that we would like to test for intersection against the bounding box. As can be clearly seen, the ray does intersect the box; we just need some way of determining this.

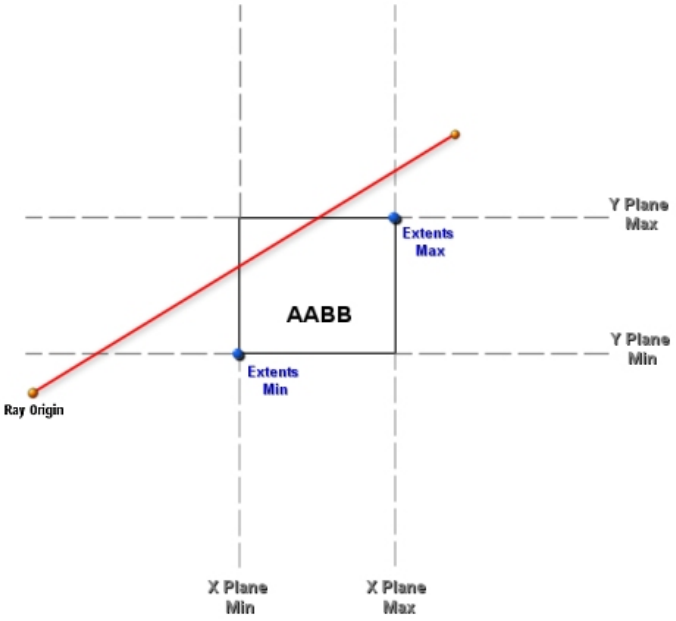


Figure 14.55

We can see that the point at the bottom left corner of the AABB (Extents Min) describes a point that is on the plane of the left face of the box. This is a plane whose normal is aligned with the coordinate system X axis. The top right corner of the box (Extents Max) describes a point that is on the same plane as the right face of the box and whose normal is also aligned to the coordinate system X axis. These two planes are parallel to one another and share the same normal, and as such, comprise a slab. This is the slab for the X axis and it contains the planes labelled 'X Plane Min' and 'X Plane Max'. They are assigned the min and max names based on their distance to the ray origin; that is, based on the time of intersection between the ray and that plane. X Plane Min for example will intersect the ray closer to the ray origin, so it is called the minimum plane of the slab. We can see that a slab bounds the AABB along one axis. In this case we are describing the X slab which bounds the AABB on its left and right sides. The reason why this is important will become clear in a moment.

If we look at the same diagram (Figure 14.55) we can see that the same two extent vectors of the AABB also describe points that are on two planes aligned with the coordinate system Y axis, and thus describe

the Y slab that bounds the box on its top and bottom sides. Thus, Extents Min is also on the Y Plane Min plane and Extents Max is also on the Y Plane Max plane. Therefore, these two extents vectors actually describe four planes and two slabs. The planes of the Y slabs would each have normals aligned with the coordinate system Y axis and are once again labelled using min and max based on their distance from ray origin (the *t* value at which the ray would intersect them).

This means that we have all the information we need to define each plane in each of the slabs. We know the plane normals used by each slab since they will be aligned with the coordinate system axis. That is, one slab will have normals <1,0,0> and the other will have planes with normals <0,1,0>. In the 3D case there will be a third slab consisting of two planes with the normal <0,0,1>. Of course, we know that a normal is not enough to describe a plane since there are an infinite number of planes that share the same normal but are positioned at different distances from the origin. However, we have the points that are on each of the four planes (the two extent vectors) so we have everything we need to test the ray for intersection against each of these four planes.

The test is very simple. For each slab, we perform an intersection test between the ray and each of the planes comprising that slab. This will return us two *t* values of intersection. For example, we might process the X slab first, which would mean performing two ray/plane intersection tests between the ray and each plane in the slab (X Plane Min and X Plane Max). Once we get back these *t* values, we store them in two variables called Min and Max, sorted by value. As we test each slab and retrieve the minimum and maximum *t* values (for each plane) we compare them against two values that are keeping a record of the highest minimum *t* value and the lowest maximum *t* values found so far. If the minimum *t* value we have just calculated for the current slab is higher than the highest minimum *t* value we have found for a previous slab, we overwrite it with the new *t* value. Likewise, if the maximum *t* value for the current slab is lower than the lowest maximum *t* value we have recorded from all previously tested slabs, we overwrite the value with this new lowest maximum *t* value.

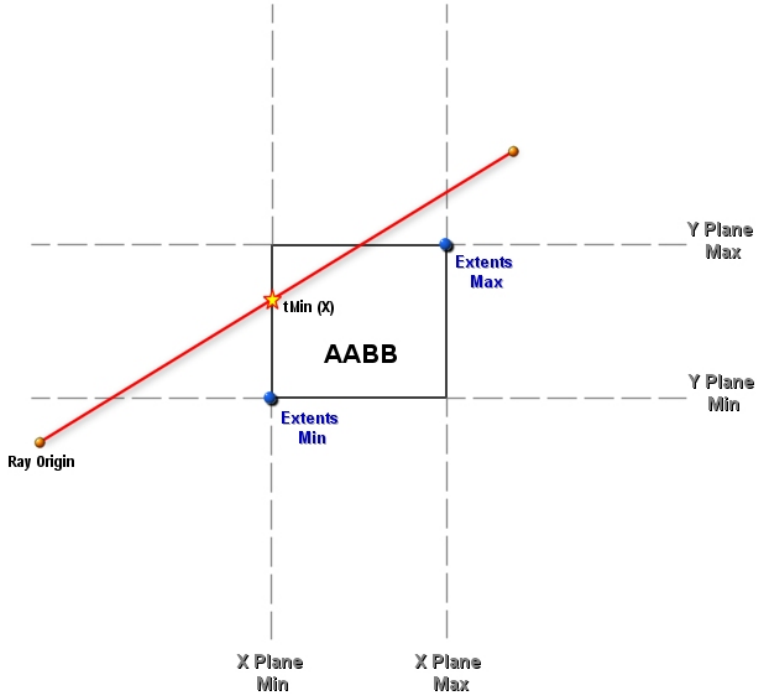


Figure 14.56

After all slabs have been evaluated, we will have two variables that contain the lowest maximum *t* value and the highest minimum *t* value we have found. If the highest minimum *t* value is larger than the lowest maximum *t* value then the ray does not intersect the box.

That might sound a little complicated, so let us have a look at an example that shows us computing the *t* values for each slab, one step at a time. In this first example, the ray does intersect the box, so the highest minimum *t* value we find for all the slabs must be lower than the lowest maximum *t* value we found from any slab according to the statement we just made a moment ago.

Let us step through the process of calculating the *t* values for the X slab first.

In Figure 14.56 we start with the X slab and calculate the *t* value of intersection between the ray and the plane that contains the minimum extents vector with a normal aligned to the coordinate system X axis (X Plane Min). The *t* value returned is shown on the diagram as tMin(x). This is the point at which the ray intersects the first plane of the X slab. Although we have already called this the minimum plane, we do not always know that this is the case. If the ray was intersecting the box from the right hand side instead, the second plane in the slab (X Plane Max) would be the first (minimum) plane of intersection.

Next we calculate the *t* value for the other plane of the slab (see Figure 14.57). This is the plane that contains the point ExtentsMax and shares the same normal (1,0,0).

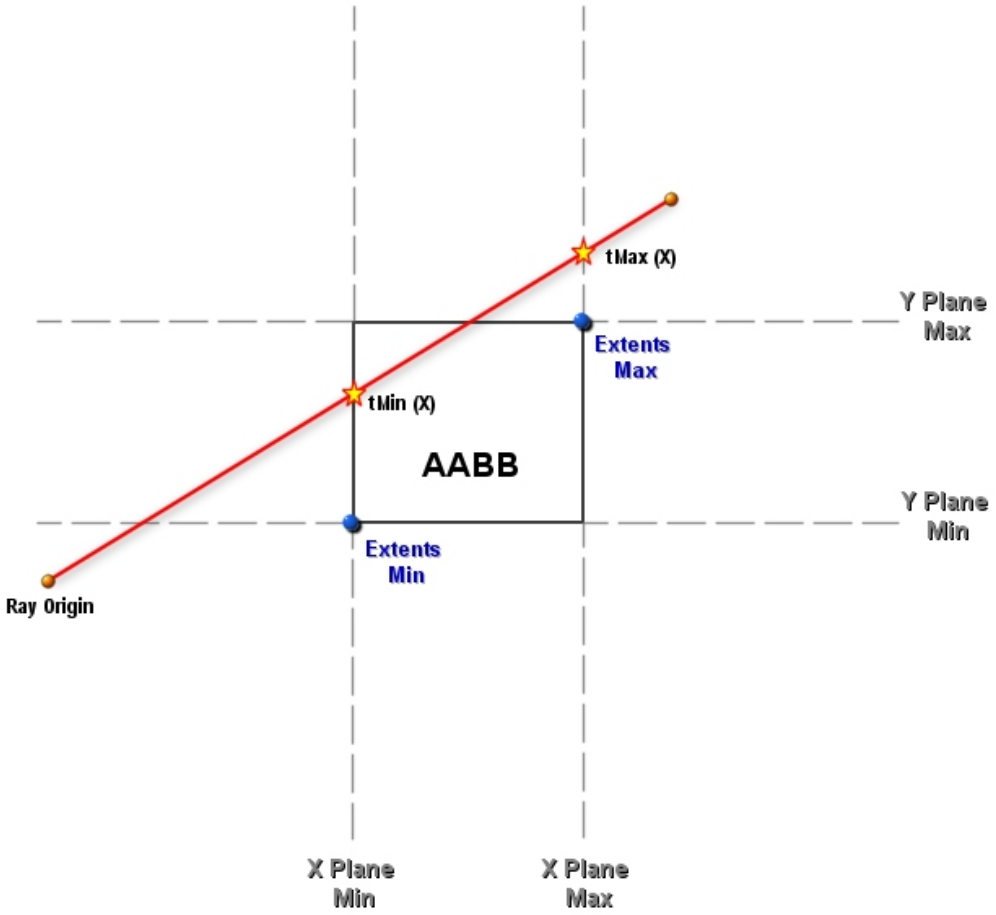


Figure 14.57

At this point we have the two *t* values for this slab, so we test which is greater and store them in the temporary variables tMin(x) and tMax(x). That is, we store in tMin(x) the plane in the slab that the ray intersects first (the smallest *t* value). In this example, we actually calculated the *t* values in this order but if the ray was intersecting from the right side of the cube, tMin(x) would contain the intersection with the X Max plane instead, and vice versa since the intersection order would be reversed.

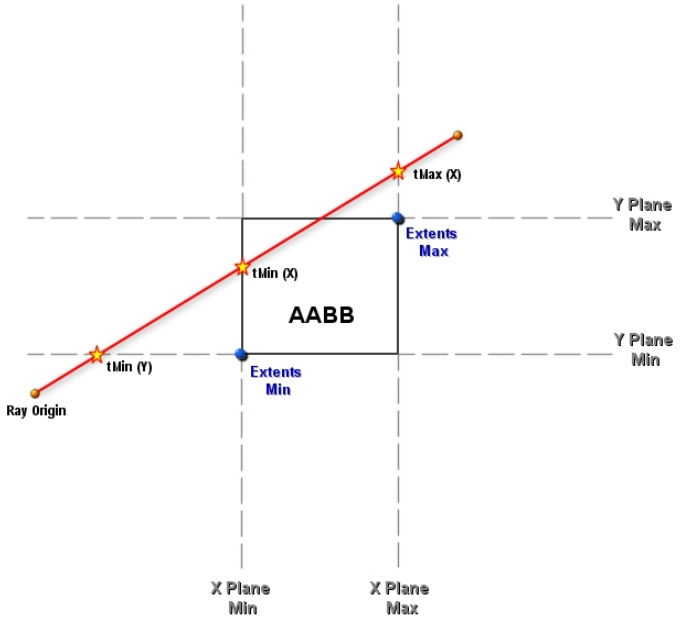


Figure 14.58

Now it is time to process the next slab (the Y slab). First we calculate the point of intersection between the ray and the first plane in the Y slab (Y Plane Min). This is shown in Figure 14.58 and is labelled tMin(y).

Our next task is to calculate another ray plane intersection test between the ray and the second plane in the Y slab (Y Plane Max). The point of intersection between the ray and second plane in the Y slab is shown in Figure 14.59 and is labelled tMax(y).

Once we have both *t* values for the Y slab, we sort them into two temporary variables as before so that tMin stores the lowest *t* value in the slab and tMax holds the highest *t* value in the slab.

The next bit of logic makes it all work. Once we have calculated the two *t* values for a slab, we see if the minimum *t* value for that slab is greater than the largest minimum *t* value we have found so far. If so, we record it. We can see in Figure 14.59 that the highest minimum *t* value from both slabs is tMin(x), which was the minimum *t* value we recorded for the first slab. In other words, this is the minimum *t* value that intersects the ray furthest from the origin compared to any other minimum *t* value calculated for other slabs. Also, when we get the maximum *t* value for a slab, we test to see if it is lower in value than any *t* value we have found so far. If it is, then we record it. At the end of testing all slabs we will have the lowest maximum *t* value found and the lowest minimum *t* value found.

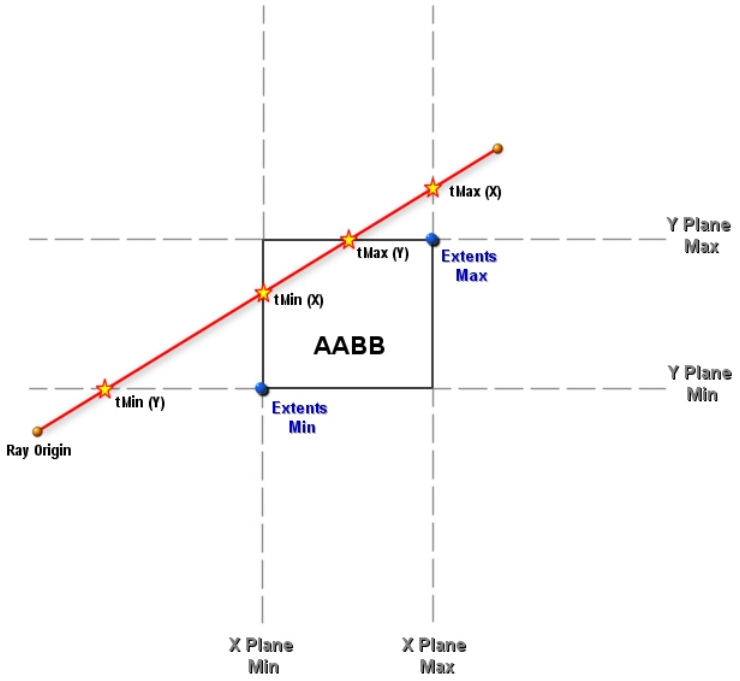


Figure 14.59

In Figure 14.59 we can see that when we calculate the maximum *t* value for the Y slab (tMax(y)), this intersection happens closer to the ray origin than the maximum *t* value found for the previous slab (yMax(x)) so we record that this is the lowest maximum *t* value we have found so far.

At the end of testing each slab we will have recorded the highest minimum value and the lowest maximum value as shown below.

tMin = tMin(X)
tMax = tMax(Y)

tMin and tMax were the temporary variables used to record the highest minimum *t* value and the lowest maximum *t* value during slab testing. We have highlighted the tMin and tMax values in Figure 14.60 below. As tMin is smaller than tMax, this means the ray intersects the box and we can return true for intersection.

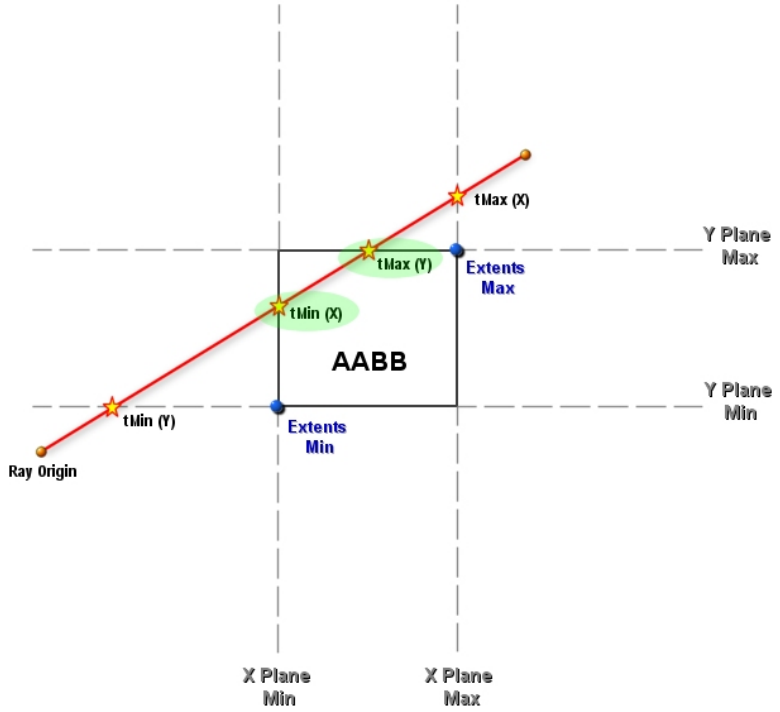


Figure 14.60

To understand why this means an intersection took place, imagine taking the red ray in the above diagram and moving it diagonally up and to the left.

As you imagine slowly moving it, you should be able to see that tMin(X) would move up its plane and tMax(Y) would move left along its plane and they would be identical values at the point where the ray just touches the top left corner. If we continue to move the ray up and to the left we can see that tMin(X) and tMax(Y) would swap around and the intersection with tMax(Y) would happen before the intersection with tMin(X). Thus we have a situation where the highest minimum *t* value is greater than the lowest maximum *t* value and thus, we have no intersection.

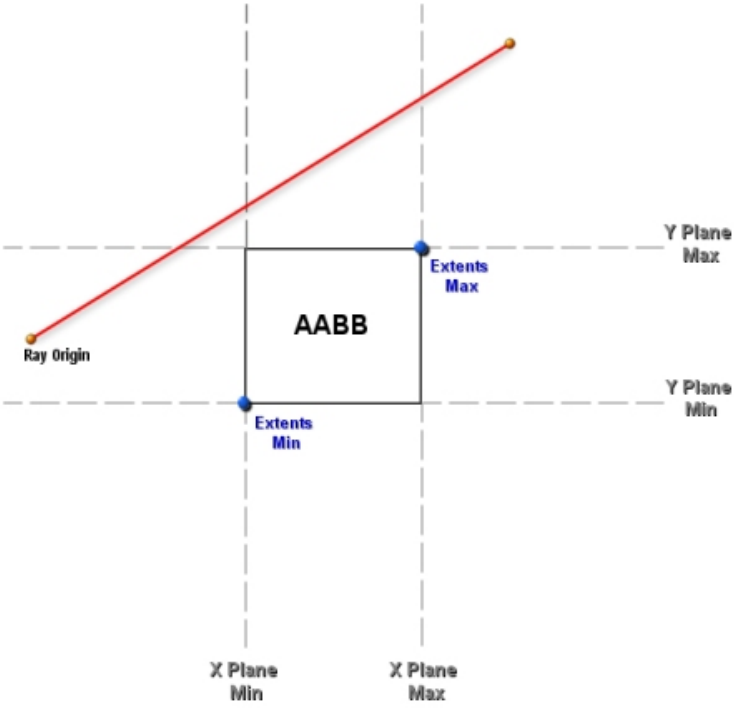


Figure 14.61

Let us try this out to make sure we are correct. Figure 14.61 shows a new example where the ray does not intersect the box. We will now quickly step through the same process and we should find that tMin is larger than tMax at the end of testing all the slabs.

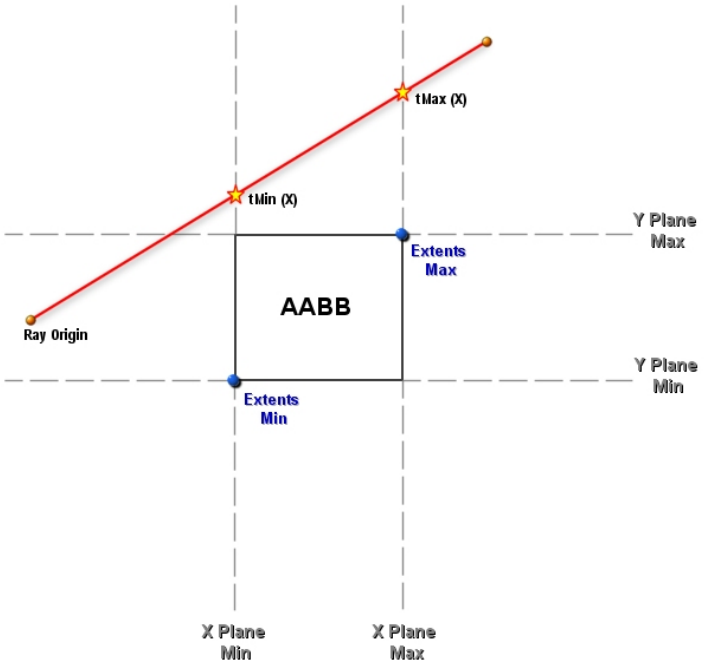


Figure 14.62

As before, we process each slab one at a time. Figure 14.62 shows the results of intersection testing the ray with the two planes comprising the X slab. Again, we test the two returned *t* values and sort them so that tMin(X) holds the first point of intersection and tMax(X) stores the second point of intersection.

Obviously, because we have only tested one slab at this point, the highest minimum *t* value we have found is tMin(X) and the lowest maximum *t* value we have found is tMax(X).

Next we move on to the Y slab and calculate the minimum and maximum intersection *t* values for both planes of that slab. These are shown in Figure 14.63 as tMin(y) and tMax(y). Notice this time however that the lowest maximum value we have found is tMax(y) which intersects the ray closer to the origin than the highest minimum *t* value (tMin(x)). As such, tMin > tMax and we know that the ray does not intersect the box. The *t* values that would end up in tMin and tMax at the end of testing all slabs are highlighted green in Figure 14.63.

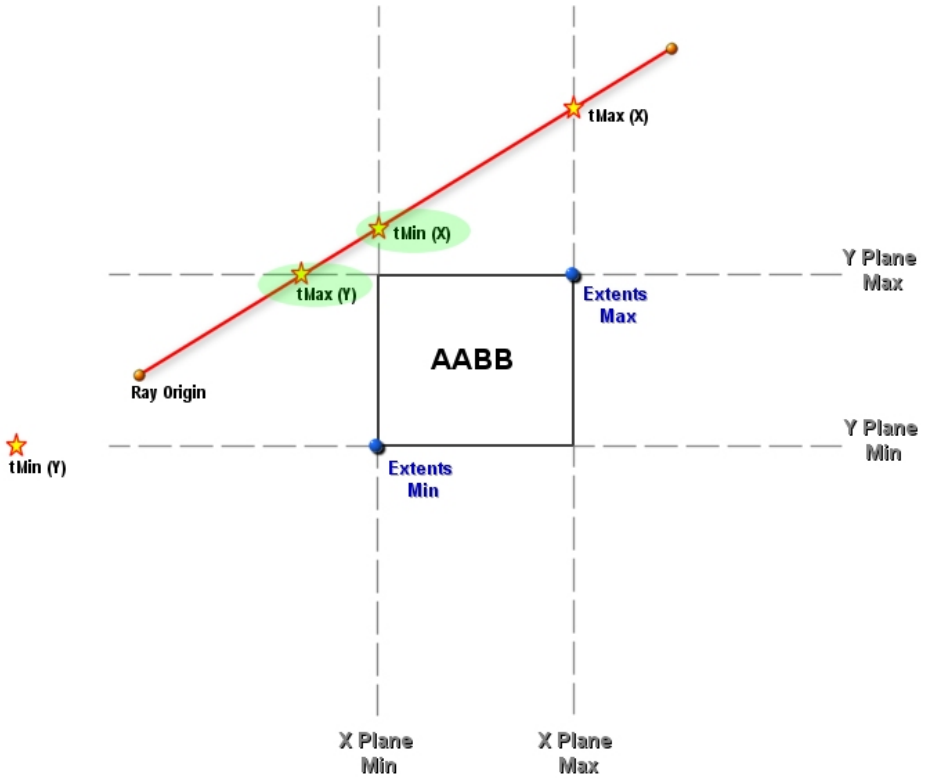


Figure 14.63

Notice that in this second example (Figure 14.63) the minimum *t* value for the Y slab (tMin(y)) is actually behind the ray origin and will thus be a negative value. This works fine since we are only interested in recording the highest minimum value found so far (so this *t* value would therefore be ignored).

Now it might not be clear why, but the process we have just described work for both OBBs and AABBs. Regardless of the orientation of the planes, as long as we have the two corner points of the box, we have points that lay on all slab planes and calculating the *t* values for each slab can be done using the CCollision::RayIntersectPlane method discussed in the previous chapter.

Next we will discuss some optimizations that can be performed that will allow our code to perform the plane intersections much quicker with axis aligned bounding boxes. Unfortunately, this will come at the cost of having a function that will not work with OBBs. However, you should be able to write your own Ray/OBB code should you need it based on the above theory.

To understand the optimizations that can be made in the case of an AABB, we will concentrate on calculating the *t* values for just one of the slabs. We will focus on the X slab for now and initially concentrate on calculating the *t* value for just one of its planes. Figure 14.64 shows the earlier example of a ray intersecting an AABB and the first plane in the X slab we need to calculate the *t* value for. As we can see, ExtentsMin describes a point on this plane and the plane normal is assumed to be aligned with the coordinate system X axis <1,0,0>.

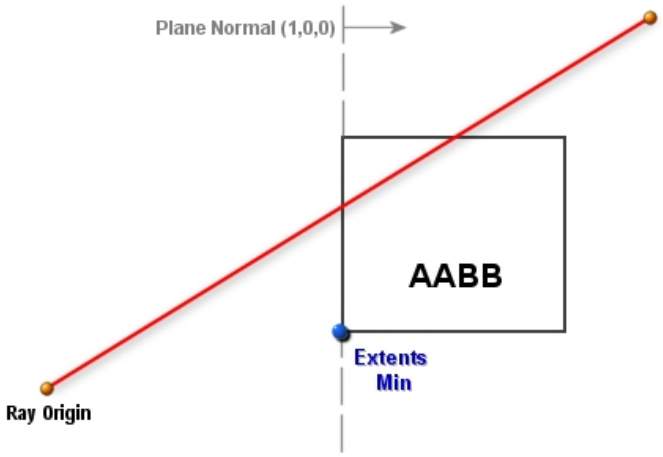


Figure 14.64

In order to find the point of intersection along the ray we first need to find the distance to the plane from the ray origin. We would normally do this by subtracting from the point known to be on the plane (ExtentsMin) from the ray origin, giving us the vector EV1 shown in Figure 14.65.

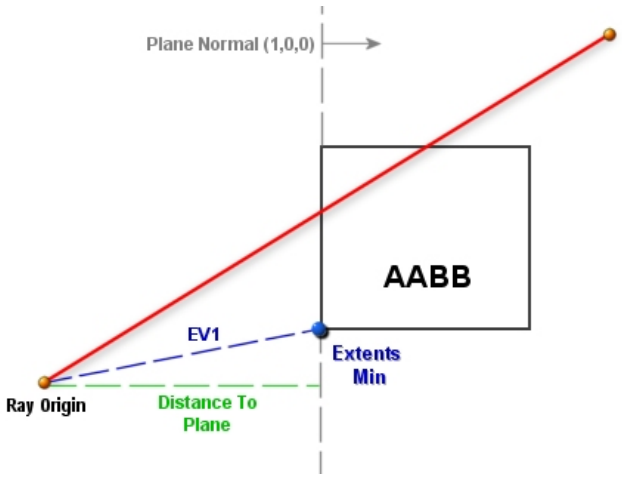


Figure 14.65

Once we have the vector EV1 we would dot it with the plane normal. Since the plane normal is unit length and the two vector origins are brought together for the dot product operation, this will scale the length of vector EV1 by the cosine of the angle between the two vectors, returning a single scalar value that describes the length of vector EV1 projected along the direction of the plane normal (the length of the green dashed line shown in Figure 14.65). This is the distance to the plane. However, why bother doing a dot product between the normal and the vector EV1 when we know that two of the components of an axis aligned plane normal will always be zero and the other will always be one? In fact, we do not have to perform a dot product between EV1 and the normal to find the distance to the plane; it is simply the value stored in EV1's x component, as shown below.

Distance To Plane **= EV1 ● Normal**
= EV1.x*Normal.x + EV1.y*Normal.y + EV1.z*Normal.z
= EV1.x*1 + EV1.y*0 + EV1.x*0
= EV1.x*1
= EV1.x

Once we have calculated vector EV1, its x component will tell us the distance to the first plane in the slab. We have just saved ourselves three multiplies and two additions per plane test.

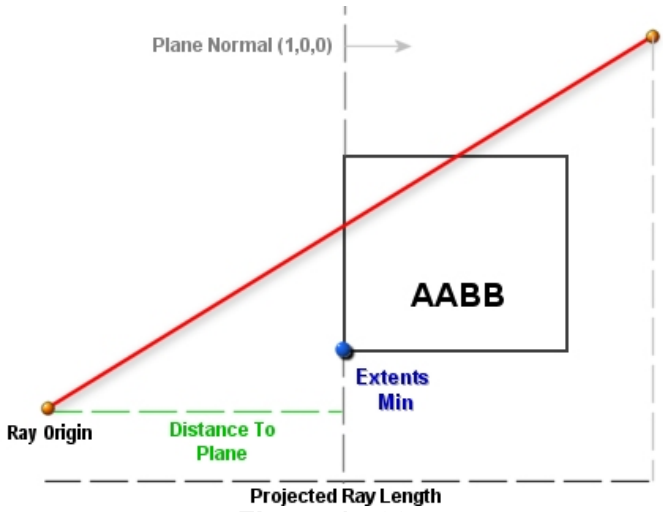


Figure 14.66

After we have found the distance from the ray origin to the plane, the next step is to project the ray length along the plane normal as shown in Figure 14.66. The ray delta vector is a vector describing the ray's direction and length from the ray origin to its end point. If we dot this with the plane normal, we will project it onto the line of the plane normal which will tell us the projected ray length. This is the length of the black dashed line at the bottom of Figure 14.66. Note that it tells us the length of the adjacent side of a triangle with the ray delta vector as the hypotenuse and the plane normal describing the direction of the adjacent side. Because we have projected the ray length along the plane normal, we can just divide the distance to the plane from the ray origin by the projected ray length to get the *t* value of intersection. In the above diagram it would be somewhere near to 0.5 as the plane intersects the ray roughly halfway along. In the last chapter we learned that we can calculate the *t* value of intersection between a ray and a plane as follows:

$$t = \frac{\text{Distance To Plane}}{\text{V} \bullet \text{N}}$$

$$t = \frac{EV1.x}{V \bullet N}$$

Where V is the ray delta vector and N is the plane normal. Once again, because we know that the plane normal of axis aligned plane will be a unit length vector with two of its components set to zero, the full dot product is not needed. This simplifies the denominator in the above equation to the following (for the X axis):

V ● N **= V.x*Normal.x + V.y*Normal.y + V.z*Normal.z**
= V.x*1 + V.y*0 + V.x*0
= V.x*1
= V.x

The final equation for calculating the *t* value for the ray and the plane shown in the above diagrams (the first plane of the X slab) has been simplified to the following:

$$t = \frac{EV1.x}{V.x}$$

We can do exactly the same for the second slab in the X plane, only this time we calculate EV1 by subtracting the ray origin from Extents Max instead of Extents Min.

This same logic works for all slabs that are axis aligned. Thus, to calculate the *t* values for the Y slab:

$$t = \frac{EV1.y}{V.y}$$

When we introduce the third dimension and a new Z aligned slab:

$$t = \frac{EV1.z}{V.z}$$

Remember that for each plane, we must calculate EV1 by subtracting the ray origin from either ExtentsMin or ExtentsMax depending on whether we are intersection testing the minimum or maximum plane of the slab.

Let us now look at the code to our new RayIntersectAABB function. The function is passed the ray origin and delta vector (Velocity) and is also passed the minimum and maximum extents of the AABB. As the fourth parameter we pass a variable by reference that will contain the *t* value of intersection between the ray and the box on function return. As with all of our AABB routines, we also offer the option to ignore any axis (slab) test. This allows us to (for example) consider a ray to intersect a box if it passes through the box in the X and Z dimensions but is above or below the box (useful when running queries against a Y-variant quad-tree).

```
bool CCollision::RayIntersectAABB(  const D3DXVECTOR3& Origin,
                                    const D3DXVECTOR3& Velocity,
                                    const D3DXVECTOR3& Min,
                                    const D3DXVECTOR3& Max,
                                    float& t,
                                    bool bIgnoreX /* = false */,
                                    bool bIgnoreY /* = false */,
                                    bool bIgnoreZ /* = false */ )
{
    float tMin  = -FLT_MAX, tMax =  FLT_MAX, t1, t2, fTemp;
    D3DXVECTOR3 ExtentsMin, ExtentsMax, P, RecipDir;
    ULONG       i;

    // Calculate required values
```

```
    ExtentsMin  = Min - Origin;
    ExtentsMax  = Max - Origin;
    RecipDir = D3DXVECTOR3( 1.0f/Velocity.x, 1.0f/Velocity.y, 1.0f/Velocity.z);
```

For efficiency we will be testing one slab at a time (two planes simultaneously), so we subtract the ray origin from both the minimum extents and maximum extents of the box. In this code, we can think of the variable ExtentsMin as describing the vector EV1 in our diagrams (the vector from the ray to the minimum plane in the slab) and we can think of ExtentsMax as describing a vector from the ray origin to the point on the second plane in the slab. At this point, the ExtentsMin <x, y, z> components contain the distances from the ray origin to the three minimum planes of each slab and ExtentsMax contains the three distances from the ray origin to the second plane in each slab. As we discussed earlier, we will calculate the *t* value by dividing the component in the ExtentsMin and ExtentsMax vectors that matches the plane we are currently testing by the matching component in the ray delta vector (Velocity). As we may have to test six planes in all, this means six divisions. Since divisions are slower than multiplications, we perform three divisions to create a reciprocal delta vector which we can then multiply with this vector. This will have the same effect as dividing by the ray's delta vector.

Now it is time to test each slab by setting up a loop to count three times. Each iteration of this loop will calculate the two t values from the planes forming that slab. As you can see below, if the slab we are currently testing is one we have chosen to ignore, we just skip any processing and advance to the next iteration.

```
    // Test box 'slabs'
    for ( i = 0; i < 3; ++i )
    {
        // Ignore this component?
        if ( i == 0 && bIgnoreX ) continue;
        else if ( i == 1 && bIgnoreY ) continue;
        else if ( i == 2 && bIgnoreZ ) continue;
```

If we get this far then this is a slab we wish to process. If we are on iteration zero then we are processing the X slab. If the matching component in the ray delta vector is zero (with tolerance) then the ray is running parallel to the current slab and could not intersect any of its planes, so we skip this slab. Therefore, the *t* value calculations for each plane in the current slab are only executed when the ray's delta vector does not have a zero in the component that matches the slab currently being tested. So if Velocity.x=0 then there is no point trying to calculate a point of intersection with the plane's comprising the X slab as the ray does not get any closer or further from the slab with distance.

```
        // Is it pointing toward?
        if ( fabsf(Velocity[i]) > 1e-3f )
        {
```

If we get inside this code block then we need to calculate the two *t* values for the planes of this slab. As discussed, this is a simple case of dividing the matching components in each of the extents vectors by the matching component in the ray delta vector.

```
            t1 = ExtentsMax[i] * RecipDir[i];
            t2 = ExtentsMin[i] * RecipDir[i];
```

At this point we have the *t* values between the ray and the planes of this slab stored in t1 and t2. We want to make sure that t1 holds the smallest *t* value, so we swap their values if this is not the case.

```
            // Reorder if necessary
            if ( t1 > t2 ) { fTemp = t1; t1 = t2; t2 = fTemp; }
```

Next we test to see if t1 (the minimum *t* value) is greater than any other minimum *t* value we have stored so far (tMin). If so, we overwrite its value with the new *t* value. Also, if the maximum *t* value (t2) is less than any maximum *t* value we have found so far (tMax), we also record its value.

```
            // Compare and validate
            if ( t1 > tMin ) tMin  = t1;
            if ( t2 < tMax ) tMax = t2;
```

At this point tMin will contain the highest minimum *t* value found so far for all previous slabs, and tMax will contain the lowest maximum *t* value. If we find (even if we are testing only the first slab) that tMin is ever greater than tMax, it means the ray cannot possibly intersect the box and we can return false immediately without testing any other slabs. Also, if we ever find that tMax is smaller that zero, then it means the box is behind the ray and we can also return false immediately. Here is the rest of the slab testing loop.

```
            if ( tMin  > tMax ) return false;
            if ( tMax < 0 ) return false;

        } // End if toward

    } // Next 'Slab'
```

If we reach this point in the function without returning then we know we have an intersection between the ray and box. All we wish to do now is return the first point of intersection. This should always be the value contained in tMin, unless it is a negative value, in which case the plane is behind the ray origin and tMax should be returned.

```
    // Pick the correct t value
    if ( tMin > 0 ) t = tMin; else t = tMax;

    // We intersected!
    return true;
}
```

That was quite a lot of explanation for what turned out to be a very small function. However, this is exactly the sort query you may have to explain at a job interview so it is good that we explored it fully.

14.15 AABB/Plane Intersection

The algorithm for classifying an AABB against a plane is not really new to us since it is something we have been doing since we first introduced frustum culling in Module I. You will recall that we introduced code to detect whether an AABB was outside a frustum plane and if so, the object which the AABB approximates was not rendered.

In this section we will cover a more generic implementation of a routine that will classify an AABB with respect to a plane and return a member of our CCollision::CLASSIFYTYPE enumeration (i.e., CLASSIFY_INFRONT, CLASSIFY_BEHIND or CLASSIFY_SPANNING). We will require a function such as this during the tree building phase where we wish to test the AABB of a detail area against the node split planes to determine which children it should be assigned to.

Classifying an AABB with respect to a plane is a simple case of finding the two corner points of the AABB that would be closest to the plane and furthest from the plane ***if the AABB was in front of the plane***. We refer to these as the *near* and *far* points. We know for example that as the near point is the closest point to the plane in a scenario where the box is in front of the plane, if this point is in front of the plane, then the whole AABB must be in the front space also. Furthermore, the far point describes the point on the box that would be farthest from the plane in the scenario (where the box was in front of the plane). If this point is behind the plane, the whole box must be behind the plane also. Finally, if the near and far points are on opposing sides of the plane, the box is spanning the plane.

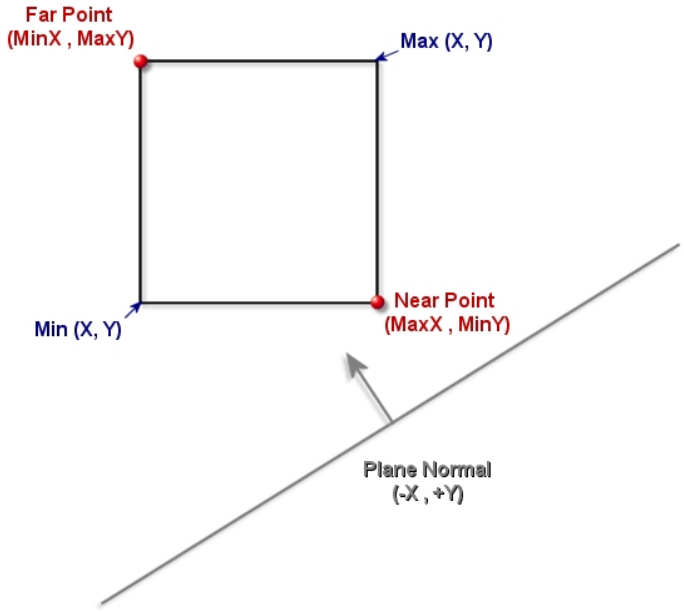


Figure 14.67

Calculating the near and far point is a simple case of analyzing the plane normal and picking the components from the AABB extents vectors to construct the point based on that normal. For example, if the x component of the normal is negative (the normal is facing in the direction of the negative X axis) then the x component of the near point will be the x component of the AABB max extents vector. If the normal is facing in the positive x direction, the x component of the AABB min extents vector describes the face of the cube that would intersect the front of the plane first were it in front of that plane. We do the same for each component of the normal. That is, for each component of the normal, if it is negative we use the maximum extents vector component for the near point and the minimum extents vector component for the far point. After doing this for each normal component, we will have constructed our two required points. In Figure 14.67 we demonstrate how this works.

Figure 14.67 depicts an AABB which is clearly in front of the plane. We also see the near and far points that have been generated based on the plane normal. Min and Max are the extent vectors of the AABB.

We can see that in this situation, if we were to move the box towards the plane, the near point would intersect the plane first and the far point would intersect the plane last. Let us examine how we generated the near and far points in this example.

The plane normal faces down the negative X axis and the Y component of the normal is positive (it is pointing up). We will keep things two dimensional for ease of illustration. Because the x component of the normal is negative, it means we use the x component from the max extents vector for the near point's x component (MaxX). Because the y component of the normal is positive, we use the y component of the AABB's minimum extents vector as the component for the near point. This gives us the point at the bottom right corner of the box shown in Figure 14.67. This is the point that would be closest to the plane if the box is in front of the plane. We can see that the reverse logic is true for the far point. If a normal component is negative, then the component from the far point is taken from the minimum extents vector of the box, otherwise it is taken from the maximum extents vector. As we can see in Figure 14.67, because the normal has a negative x component, we use MinX for its x component. Because the y component of the plane normal is positive, we take the y component from MaxY. This gives us the far point shown. Because the near point is in front of the plane, the entire box must be in front of the plane also, so we can return CLASSIFY_INFRONT.

It is important to realize that the near and far points are always calculated in this way regardless of the orientation of the plane or the position of the box. Therefore, the near point is not always the point that is nearest to the plane; it is the point that *would* be nearest to the plane *if the box was in front of the plane*.

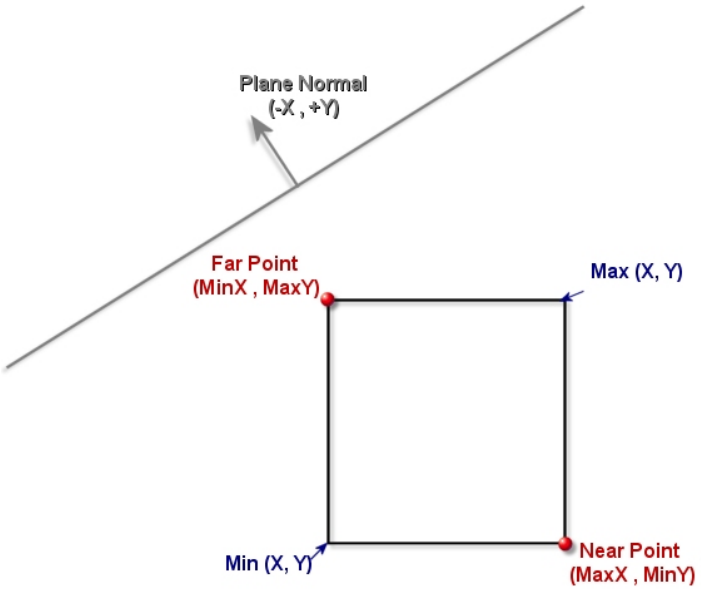


Figure 14.68

In Figure 14.68 we show the same plane being used but we have moved the box so that it is now behind the plane. Since the plane normal is the same, the near and far points of the box are calculated in exactly the same way. However, notice that because the box is behind the plane, the near point is no longer the point that is closest to the plane. We can see in this example that as the far point would be the last position on the cube to pass the plane if the cube was being moved from the planes front halfspace to its back halfspace, if the far point is behind the plane, the entire AABB must be behind the plane also and we can return CLASSIFY_BEHIND.

Of course, it is also clear to see when looking at Figures 14.67 and 14.68 that if, after calculating the near and far points, we find that they are on different sides of the plane, the box must be spanning the plane and we can return CLASSIFY_SPANNING.

The spanning case is illustrated in Figure 14.69. It demonstrates the fact that because we are now using a different plane normal, the near and far points are calculated completely differently. In this example,

because the plane normal has a negative x and y component, the near point is simply the AABB's minimum extents vector and the far point is the AABB's maximum extents vector.

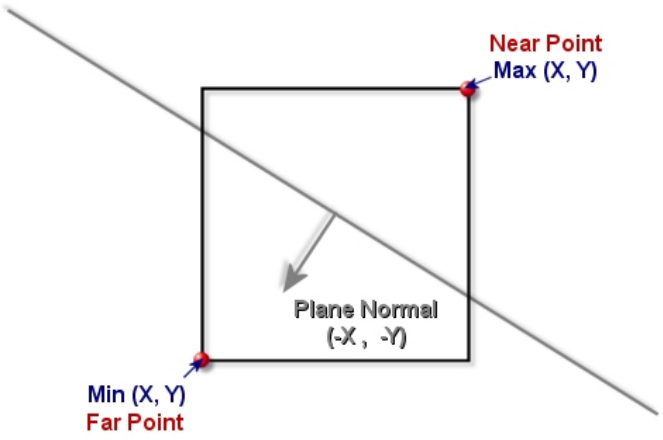


Figure 14.69

Below we see the code to the CCollision::AABBClassifyPlane function which implements the steps we have just discussed. Its parameters are the extent vectors of the AABB and the plane. The plane is passed in using ABCD format (a normal, ABC, and a distance from the plane to the origin of the coordinate system, D).

After calculating the near and far points, we classify the near point against the plane. If it is in front of the plane we can return CLASSIFY_INFRONT immediately since the whole box must be in front of the plane as well. If not, it means the near point is behind the plane so we test the far point. If the far point is in front of the plane, it means the box must be spanning the plane because the near and far points are in different half spaces. If this is the case, we return CLASSIFY_SPANNING. If not, the box must be behind the plane, so we can return CLASSIFY_BEHIND.

```
CCollision::CLASSIFYTYPE CCollision::AABBClassifyPlane( const D3DXVECTOR3& Min,
                                                        const D3DXVECTOR3& Max,
                                                  const D3DXVECTOR3& PlaneNormal,
                                                        float PlaneDistance )
{
    D3DXVECTOR3 NearPoint, FarPoint;

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

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

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

    // If near extreme point is outside, then the AABB is totally outside the plane
    if ( D3DXVec3Dot( &PlaneNormal, &NearPoint ) + PlaneDistance > 0.0f )
         return CLASSIFY_INFRONT;
```

```
    // If far extreme point is outside, then the AABB is intersecting the plane
    if ( D3DXVec3Dot( &PlaneNormal, &FarPoint ) + PlaneDistance > 0.0f )
         return CLASSIFY_SPANNING;

    // We're behind
    return CLASSIFY_BEHIND;
}
```

Yet another function has now been added to our intersection library that we will find ourselves using time and time again as we progress with our studies. This concludes our discussion of the new intersection and classification routines we need to add to our library in order to build spatial hierarchies.

14.16 T-Junctions

Earlier in the lesson, we learned that any extensive clipping procedure will likely introduce T-junctions in the geometry. In this section we will examine what T-junctions are, the displeasing visual artifacts they exhibit, and a way to fix them. Since all of our trees will have the ability to produce clipped trees, all of our derived tree classes will want to have their T-junctions fixed after the build process has completed (if the tree is intended to be used for rendering). Since fixing T-junctions is exactly the same algorithm regardless of the tree type being used (it is just a repair pass on the polygons in the tree's polygon list), we have decided to place the T-junction removal code inside the Repair method of CBaseTree. This way, the same function can be used to fix the geometry built by any of our derived tree classes.

As T-junctions exhibit themselves only when the data that contains the T-junctions is being rendered, the derived class (or the application) will not have to issue the Repair call after the tree has been built as it will happen automatically. That is, the CBaseTree::PostBuild method, which is called from the derived class Build function after the tree has been built, will issue a call to the CBaseTree::BuildRenderData method. Although this method will be discussed in the following lesson, it essentially prepares the tree polygon data so that it is ready to be rendered. This includes making a call to the CBaseTree::Repair function. Of course, the BuildRenderData method returns immediately if the tree has not been given a valid 3D device pointer which means the CBaseTree::Repair method will only be called automatically for trees that you intend to render. If you are not using the tree for rendering (just collision queries) and you have not passed a 3D device pointer into the tree's constructor (avoiding the render data being built), but would still like to have the T-junctions repaired, your application can issue a call to the Repair method after the Build method has returned. Once again, remember that if a device pointer was passed to the constructor, the render data would have been built and the T-junction repair function called automatically.

14.16.1 What are T-Junctions?

A T-Junction is the meeting of geometry along a shared edge where the vertices along that shared edge are not equal in each object. This can happen a lot when clipping is being performed. A quad for example might be split into two pieces vertically at one node, and then one of those splits later gets split into two pieces horizontally at another node (see Figure 14.70). We can see in this example that the polygons (which may have originally been one quad before clipping was performed) all share a common edge. That is, the right edge of the left polygon would be butted up against the left edges of the two smaller polygons to its right. This is a classic T-junction scenario.

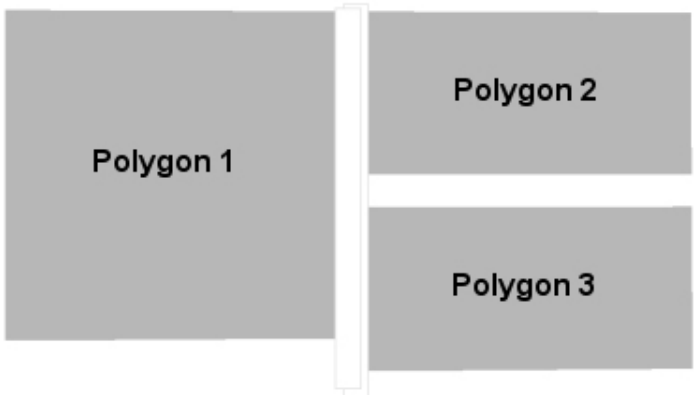


Figure 14.70

In this image we have deliberately separated the three neighboring polygons so that you can clearly see the polygon boundaries, but you should be able to recognize that if we were to slot all three polygons together so that there were no longer any gaps between them, they would look like a single large quad. Perhaps these three polygons were a single quad to begin with, but clipping broke it into three pieces. Nevertheless, we still want it to look like a single quad when rendered.

Of course, this same situation may arise simply because the artist has designed the geometry that way instead of it having been introduced by a clipping process. Either way, we have ourselves a T-junction to deal with and our scene may contain many of them.

While looking at the above image, if you tilt your head to the left, you should be able to see that the white gaps between the polygons form the letter ‘T’, which is where such a geometric configuration gets its name. If these polygons are neighbors (i.e., if their edges are touching) we would certainly have a problem that would cause lighting anomalies and other visual artifacts. In Figure 14.71 we have connected the polygons together to better show the problem. The polygon boundaries causing the T-junction are highlighted with red dashed lines.

Recall that a vertex lighting system using gouraud shading records lighting samples at the vertices for interpolation over the surface. We can see that while Polygon 1 shares its right hand edge with both Polygons 2 and 3, Polygons 2 and 3 have a vertex (shown as the red sphere) in the center of polygon 1’s right hand edge. But Polygon 1 does not have a vertex in that position since its vertices are the four corner points of the quad.

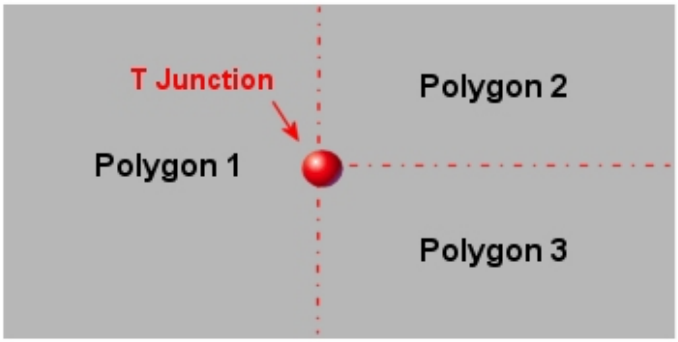


Figure 14.71

The problem occurs when a light source affects the center point of this construct where these three polygons meet. If we used a tightly focused beam (such as a small spot light for example), we would fully expect the influence of the light to affect all three polygons in the same way. However, as Figure 14.72 clearly shows, if we shine a small spot light at the center point of this construct, the bottom left vertex of Polygon 2 and the top left vertex of Polygon 3 are very close to the light source and as such, a high intensity light sample is recorded at those vertices. If Polygon 1 had a vertex in the same position, then it too would receive that same light sample which would be interpolated over its entire surface. The problem is that it does not have a vertex located there. In fact, Polygon 1's vertices are some distance from the light source. So although this construct of three polygons might seem to the user to be a large single wall polygon (for example), we have the lighting suddenly cut off as the light passes into Polygon 1's area because it does not have a vertex in a correct position to sample the light in the same way as its neighbors. This causes the lighting discontinuity illustrated in Figure 14.72.

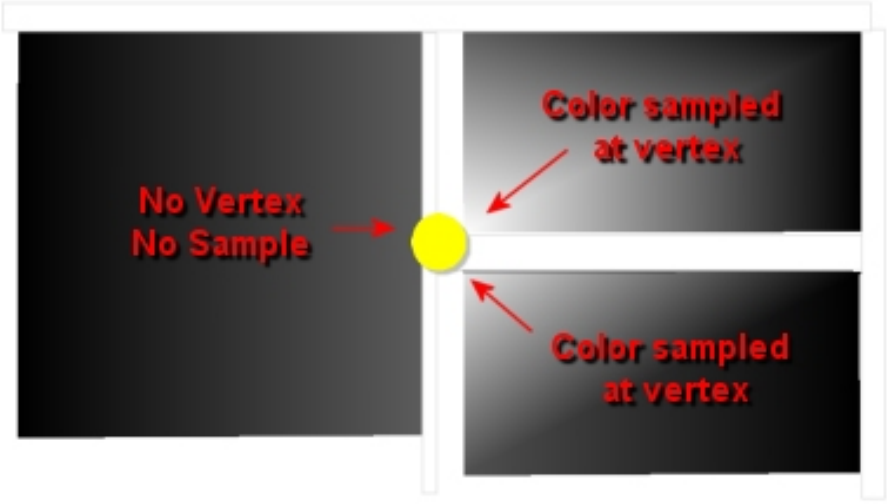


Figure 14.72

In Figure 14.73 we see the result is made much worse if the polygons are connected as they are supposed to be in this example. This is an obvious lighting discontinuity that would stick out like a sore thumb to even the most casual gamer.

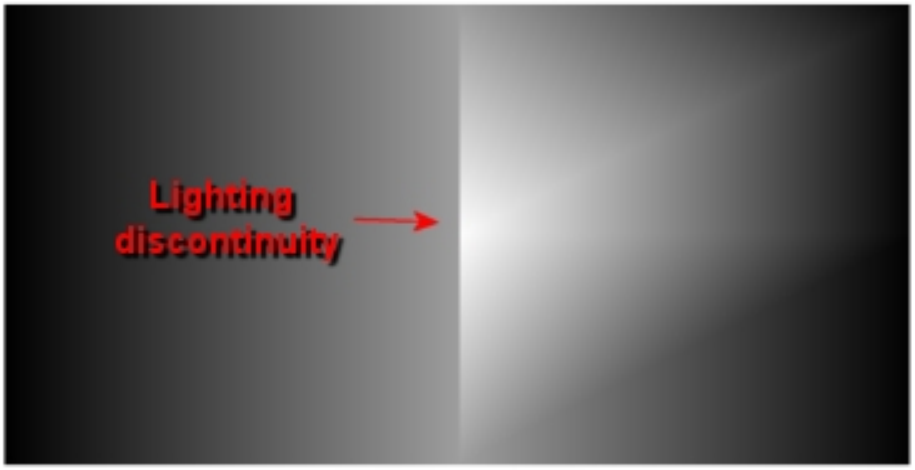


Figure 14.73

Whenever a T-junction occurs within a game level, we have the potential for a vertex lighting system to suffer these visual glitches. In fact, T-junctions are not just undesirable for a vertex lighting system; they also cause visual anomalies during rasterization. Many of you may have seen T-junctions in commercial games that did not even use a vertex lighting system. They can manifest themselves in the rasterization phase as a phenomenon called “sparklies”. Sparklies are sub-pixel gabs that appear when polygons that share edges with T-junctions are being rendered. In a T-junction scenario, rounding errors caused during rasterization occur differently along the edge that is missing the vertex versus the shared edges that have the additional vertex. The polygons appear to come apart at the shared edge ever so slightly at certain pixels, causing a gap. The difference in floating point rounding errors between the edges that do and do not contain the vertex causes a situation where neither edge renders a pixel into the frame buffer, thus allowing the background to show through. These gaps are often seen as a sparkle effect as lots of little oddly colored pixels shimmer in and out of existence on the display. These colored pixels are actually the colors of more distant polygons or the color that the frame buffer was initially cleared to before the frame was rendered.

In Figure 14.74 we see a screenshot from Lab Project 14.1 with the T-junction repair step turned off. Although it is hard to take a good screenshot of sparklies (they look much worse when the camera is actually moving), even in this image we can see some cyan colored pixels (the color the frame buffer was cleared to before the scene was rendered) showing through sub-pixel gaps on the floor.

Figure 14.74

When the player is actually walking along the floor and the level is fully animated, many of these little blue dots shimmer in and out of existence. This is very distracting to the player to say the least. In fact, it is for this reason, more so than the lighting discontinuities, that we must fix these T-junctions before the scene is rendered. Since we will be moving away from vertex lighting systems in Module III of this series and begin to favor more advanced techniques, T-junctions will no longer affect our lighting system. However, these sub-pixel gaps that occur during the rasterization of shared edges that contain T-junctions are clearly unacceptable.

As mentioned, T-junctions are not just caused by clipping geometry. They can be very easily introduced by the artist during level design. T-junctions can even exist between neighboring objects/meshes that will cause lighting anomalies (see Figure 14.75).

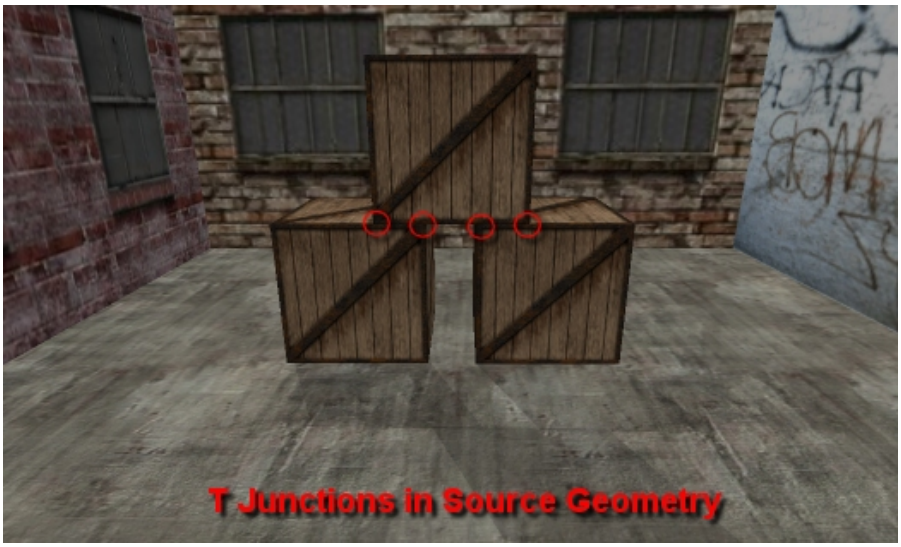


Figure 14.75

In Figure 14.75 we see how the project artist might innocently place three crates on top of each other in a staggered fashion. Because these are separate meshes, we will not have the sparklies problem, but this would still cause lighting discontinuities if a vertex lighting system was being employed.

Assuming the front face of each create is a single quad with the vertices in its four corners, we can see that top right vertex of the bottom left crate and the top left vertex of the bottom right crate partially share an edge with the bottom edge of the top crate.

Because the crates are staggered, the vertices in the bottom two crates that share the edge with the top crate have vertices in different positions versus those of the bottom edge of the top crate (and vice versa). We can also see that the bottom left and right vertices of the top crates are positioned in the middle of the top edges of the bottom crates and as such, we have four vertices that share an edge which do not each belong to both the polygons that share the edge. Take a look at Figure 14.76 and see how the vertex lighting system would light these crates if two tightly focused spot lights were set to illuminate the top left vertex of the bottom right crate and the top right vertex of the bottom left crate.

Figure 14.76

As you can see, the tightly focused beam illuminates only the points in space where the top right vertex of the bottom left crate and the top left vertex of the bottom right crate are located. The lighting color is sampled at this point and interpolated over the surfaces of the bottom two crates. However, the bottom edge of the top crate does not have vertices in these locales and its vertices (the bottom left and bottom right vertices of its quad) fall just outside the influence of the light source. As a result, the topmost crate has no vertices located in positions that receive light contributions and the create remains unlit. This has caused a very unnatural lighting result.

14.16.2 Fixing T-Junctions

Fixing T-junctions in our level is actually much simpler than you might assume. We essentially need to test the edges of each polygon in our scene, with the edges of every other polygon in the scene, searching for vertices in the edge of one polygon that exist along the edge of the first polygon. If such a vertex is found, we make a copy of the vertex and insert it into the polygon edge that has missing the vertex. Essentially, we are just finding any vertices in any polygon that may live in the middle of an edge and if found, we insert that vertex into the edge. The insertion of this extra vertex introduces another triangle primitive that will be needed to render the N-gon, so our primitive count will have grown after the T-junction repair process is complete. The basic pseudo code (lacking any optimizations) is:

```
For Each Polygon (PolygonA)
        For Each Edge (EdgeA)
                For Each Polygon (PolygonB)
                        For Each Vertex (VertexB)

                                D= Calc Distance from VertexB to EdgeA
                                ( If D !=0 [with epsilon] ) continue;

                                If  (VertexB is not between EdgeA.vertex1 and EdgeA.vertex2) continue;

                                // Vertex B is causing T junc with this EdgeA so inside it into polygon
                                PolygonA.InsertVertex( just after Edge1.vertex1);

                        End For Each Vertex
                End For Each Polygon
        End For Each Edge
End For Each Polygon
```

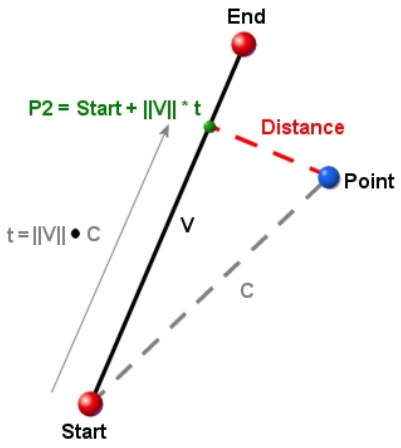
As you can see, we process each polygon and its edges one at a time. For each edge we test the vertices of every other polygon in the scene to see if that vertex causes a T-junction along the current edge. If it does, then we need to insert that vertex into the current edge so that both polygons have vertices in the same places, thus repairing the T junction. The key here is finding out whether the vertex we are currently testing is on the current edge we are testing because if it is not, it cannot possibly cause the T-junction with the current edge and we can skip it.

You can see in the above pseudo-code that for each vertex we calculate D, the shortest distance from the vertex to the infinite line on which the edge lives. If this is not zero (with tolerance) then the vertex cannot be on the edge. Obviously, if a point is on an edge, the distance from that point to the edge would be zero. If it is zero then we know that the vertex lies on the infinite line on which the edge/ray exists, but we do not know whether it lies within the line segment defined by the edge's two vertices. If it does not, then the vertex is not sharing an edge and we can ignore it. If it is between the two vertex positions then we definitely have found a vertex from another polygon that is on the edge we are currently testing in a position where the current edge does not have a vertex. When this is the case, we repair the T-junction by inserting this vertex into the edge just after the edge start vertex. This creates a new edge and a new triangle in the polygon and fixes the problem. Of course, when we insert this new vertex into the

polygon edge we must also generate its normal and texture coordinate. We can do this using interpolation based on its position between the two original vertices of the edge. Notice that testing Polygon A against Polygon B is not enough, we must also test Polygon B against Polygon A performing the same logic. The above algorithm will make sure this happens. The loops ensure that every polygon will be tested against every other polygon, so we will do ultimately perform n^2 tests (where n is the number of polygons).

It should be clear looking at the above code that the heart of this process is simply determining whether the vertex of a polygon lies on the edge of another. This process is reliant on us being able to calculate the distance from a vertex to a line segment (the edge). So we will need a function that will not only tell us the distance from a vertex to an infinite line, but will also inform us whether the point is inside the vertices of the line segment (the edge) assuming this distance is zero. Only if the distance from the vertex to the line is zero and if the point is positioned between the two edge vertices (contained within the line segment) do we wish our function to return a valid distance. So let us have a look at how we might calculate the distance from a point to a line segment.

14.16.3 Distance from a Point to a Line Segment



Calculating the Distance
from a Point to a Line Segment/Ray

Figure 14.77

Figure 14.77 illustrates the problem we are trying to solve. The blue sphere labeled Point is an arbitrary point that we would like to calculate the distance for. The line segment (the polygon edge in our example) has its two vertices shown as the red spheres labeled Start and End. The length of the red dashed line labeled Distance is what we are trying to find. If the distance is zero, then the point must lay on the same line as the edge. It may or may not be between the two vertices, but we will worry about that in a moment. For now, we are just interested in calculating the distance from the point to the infinite line on which the edge is contained.

What we are actually trying to do is calculate the position of the small green sphere in the diagram. This vector describes the projection of the point onto the line and thus, by subtracting this position from the original position, we get a vector matching the red dashed line in the diagram. The length of this vector is the distance we are seeking.

Finding the position of the green sphere is easy given the projection properties of the dot product. We first subtract the start of the line segment (the first vertex in the edge) from the point in question. This gives us vector C in the diagram. The length of this vector would tell us the distance from the origin of the ray/line segment to the point. Vector V in the diagram is calculated by the subtracting the edge start from the edge end position. V is essentially the ray delta vector with the start of the edge as the ray origin. If we normalize vector V, we can perform the dot product between unit length V and vector C to get t, the distance to travel in the direction of V from the ray start point to reach the green sphere.

All we have done is projected the length of vector C along the direction of V, which we know scaled it by the cosine of the angle between them. *t* now tells us how far we would have to travel along V from the start point to reach the green sphere. Therefore, we can calculate the position of this sphere simply by scaling unit length V by *t* and adding the resulting scaled V vector to the start of the edge. Now we have the position vector of the green sphere which we subtract from the position vector of the original point, which gives us the vector shown as the red dashed line in the diagram. We can then just return the length of this vector.

However, we also want our function to let the caller know when the point does exist on the infinite line for the edge, but is outside the boundaries of the edge vertices. That is no problem because we will know this as soon as we calculate *t*. Since V is unit length at this point, we know that *t* will be equal to zero at the edge start point and the pre-normalized length of V at the edge end point. Thus, as soon as we calculate *t*, we will return the highest possible distance value (FLT_MAX) if *t* is smaller than zero or greater than the original length of the edge. As our T-junction code will only take any action if the distance returned is zero, the function will only return zero if the point is indeed on the edge (and not just the infinite line on which the edge lies).

Next we see the code to the function. Because this is not a collision function, and would not really fit well in our CCollision namespace, we decided to add such a math utility function to a new cpp file called MathUtility.cpp. As we progress with our studies, we will place any functions we write which are general math utilities in this file. This function is contained in the MathUtility namespace defined in MathUtility.h.

MathUtility.h

```
namespace MathUtility
{
    float DistanceToLineSegment (   const D3DXVECTOR3& vecPoint,
                                    const D3DXVECTOR3& vecStart,
                                    const D3DXVECTOR3& vecEnd );
};
```

MathUtility.cpp – DistanceToLineSegment function code

```
float MathUtility::DistanceToLineSegment(       const D3DXVECTOR3& vecPoint,
                                                const D3DXVECTOR3& vecStart,
                                                const D3DXVECTOR3& vecEnd )
{
    D3DXVECTOR3 c, v;
    float       d, t;

    c = vecPoint  - vecStart;
    v = vecEnd - vecStart;
    d = D3DXVec3Length( &v );

    // Normalize V
    v /= d;

    // Calculate final t value
    t = D3DXVec3Dot( &v, &c );

    // Check to see if 't' is beyond the extents of the line segment
```

```
    if (t < 0.01f)     return FLT_MAX;
    if (t > d - 0.01f) return FLT_MAX;

    // Calculate intersection point on the line
    v.x = vecStart.x + (v.x * t);
    v.y = vecStart.y + (v.y * t);
    v.z = vecStart.z + (v.z * t);

    // Return the length
    return D3DXVec3Length( &(vecPoint - v) );
}
```

The function above mirrors Figure 14.77. It is passed the two position vectors of the line segment (the start and end points) and the point we wish to classify. We first calculate c and v as discussed, then store the length of the line segment (edge) in d. We will need this value later and we are just about to normalize it. We then normalize vector V by dividing it by its length and calculate *t* by dotting unit length vector v with c. This projects the length of c onto v (scaled by the cosine between them), returning a *t* value in the range [0, d] for points that are contained between the two end points.

If *t* is less than zero or larger than the length of the line segment, then it does not matter what the distance to the infinite line is; this point could not possibly be contained in the line segment. Its projection places it on the infinite line either prior to the start vertex or after the end vertex. When this is the case we return a distance of FLT_MAX (the maximum value that can be stored in a float). If this is not the case, then we know that the projection of the point onto the line segment did indeed produce a point (the green sphere in Figure 14.77) that is on the edge. However, we have no idea how far it is from that edge; only if it is at a distance of zero is the point truly on the edge.

Next we calculate the position of the green sphere in Figure 14.77 and store the result in v. We calculate it by scaling v by *t* and adding the result to the start vector of the edge. Now that we have the projected position stored in v, we can simply subtract it from the position of the original point and return the length of the resulting vector. If this length is zero, the point is on the edge and we have ourselves a T-junction that needs repair.

14.16.4 The T-Junction Repair Code

We now have everything we need to write our T-junction repair code. However, when we examine the pseudo-code discussed previously, we realize immediately that it will be an incredibly slow operation. In that vanilla description of the algorithm, we had to test each edge of every polygon against every vertex in every other polygon. Since most of us will be reluctant to wait for an extended amount of time for our T-junctions to be repaired, we need to speed this up as much as possible.

As it happens, by the time the repair process is invoked, the spatial tree will have already been built. Therefore we can reduce the time it takes to compute this process down to a mere fraction of the time it would otherwise take if we incorporate a spatial hierarchy into the process. At the point the repair process is called, each polygon will have already been assigned to its leaves and each polygon will have

a bounding box of its own. Therefore, much like we will implement the broad phase collision step, we will only have to test a mere handful of vertices during the repair process for each polygon's edge.

For each polygon, we will call the CollectLeavesAABB function which will be discussed in detail shortly. This is the function we will implement in our derived classes that will traverse the tree and return any leaves whose AABB intersects the query AABB. In this instance, the query AABB will be the AABB of the polygon we are currently repairing. The CollectLeavesAABB function will return a list of all leaves that intersect the polygon's AABB, allowing us to reject virtually every other polygon in the scene from consideration. Only the polygons in the returned leaves will need to be tested against each other. These are the polygons that are in close proximity to the polygon currently being repaired and might possibly share an edge with it. We will then loop through each polygon in the returned leaves and compare the bounding box of the polygon with the bounding box of the polygon we are currently repairing. Only when the two bounding boxes intersect will we test each vertex in that polygon against each edge in the polygon currently being repaired.

Repair - CBaseTree

The CBaseTree::Repair function is actually a wrapper around the core T-junction repair process. It takes care of looping through each polygon and collecting the leaves that intersect its bounding box. It also takes care of comparing the AABBs of the two polygons to determine whether a T-junction repair test should be done on them. Once it finds two polygons that have intersecting AABBs (and thus the potential to share an edge), it calls the CBaseTree::RepairTJunctions function to actually repair it. We will look at this method after we have examined the code to the main Repair function. This code is an excellent example of how hierarchies can speed up virtually any task done at the scene level.

In the first section of the code we set up a loop to iterate through all the CPolygon pointers stored in the tree's m_Polygons linked list. Remember, this list will contain every polygon being used by the tree. For each polygon in the list we will then extract its AABB. This is our current polygon being tested for repair.

```
bool CBaseTree::Repair( )
{
    ULONG                    SrcCounter, DestCounter, i;
    CPolygon                *pCurrentPoly, *pTestPoly;
    PolygonList::iterator    SrcIterator, DestIterator;
    LeafList::iterator       LeafIterator;
    LeafList                 LeafList;

    // No-Op ?
    if (m_Polygons.size() == 0 ) return true;

    try
    {
        // Loop through Faces
        for ( SrcCounter = 0,
              SrcIterator = m_Polygons.begin();
              SrcIterator != m_Polygons.end();
              ++SrcCounter,
```

```
              ++SrcIterator )
        {
            // Get poly pointer and bounds references for easy access
            pCurrentPoly = *SrcIterator;
            if ( !pCurrentPoly ) continue;

            D3DXVECTOR3 & SrcMin = pCurrentPoly->m_vecBoundsMin;
            D3DXVECTOR3 & SrcMax = pCurrentPoly->m_vecBoundsMax;
```

At the head of the function we declared a local variable LeafList, which is a variable of an STL list that stores ILeaf pointers. We make sure this list is empty and then send this list along with the AABB of the current polygon into the CollectLeavesAABB method.

```
            // Retrieve the list of leaves intersected by this poly
            LeafList.clear();
            CollectLeavesAABB( LeafList, SrcMin, SrcMax );
```

The CollectLeavesAABB code will be discussed in a moment since it is implemented in the derived classes. The method fills the passed leaf list with any leaves in the tree which intersect the bounding box of the polygon. On function return, LeafList will contain a list of ILeaf pointers which contain polygons that have the potential to cause T-junctions with the current polygon (i.e., a broad phase).

We will now loop through each leaf in the returned list and extract its pointer so that we can access its data. Notice how LeafList is defined to store ILeaf pointers, but we know that they will contain pointers to CBaseLeaf objects because this is the type of leaf we will allocate and store during the build process. Therefore, we cast the current ILeaf pointer to a CBaseLeaf.

```
            // Loop through each leaf in the set
            for ( LeafIterator = LeafList.begin();
                  LeafIterator != LeafList.end(); ++LeafIterator )
            {
                CBaseLeaf * pLeaf = (CBaseLeaf*)(*LeafIterator);
                if ( !pLeaf ) continue;
```

For the current leaf, we will set up a loop that allows us to retrieve each of its polygon pointers one at a time and perform an AABB/AABB test between it and the current polygon being repaired. Notice how we use our new CCollision::AABBIntersectAABB function to perform the AABB test between the two polygons. If it returns true, the polygon do have intersecting bounding volumes and we will have to feed them both into CBaseTree::RepairTJunctions where they will be tested at the edge/vertex level. The remaining code of the Repair method is shown below.

```
                // Loop through Faces
                DestCounter = pLeaf->GetPolygonCount();

                for ( i = 0; i < DestCounter; ++i )
                {
                    // Get poly pointer and bounds references for easy access
                    pTestPoly = pLeaf->GetPolygon( i );
                    if ( !pTestPoly || pTestPoly == pCurrentPoly ) continue;
                    D3DXVECTOR3 & DestMin = pTestPoly->m_vecBoundsMin;
                    D3DXVECTOR3 & DestMax = pTestPoly->m_vecBoundsMax;
```

```
                    // Do polys intersect
                    if ( !CCollision::AABBIntersectAABB( SrcMin,
                                                         SrcMax,
                                                         DestMin,
                                                         DestMax ) ) continue;

                    // Repair against the testing poly
                    RepairTJunctions( pCurrentPoly, pTestPoly );


                } // Next Test Face

            } // Next Leaf

        } // Next Current Face

    } // End Try Block

    catch ( ... )
    {
        // Clean up and return (failure)
        return false;

    } // End Catch Block

    // Success!
    return true;
}
```

So we have seen that While the CBaseTree::Repair method is called post-build to repair T-junctions, the actual function code really just implements a broad phase testing process for the real T-junction repair code. Our collision system will use the hierarchy in a very similar way. The only polygons we have to test at the edge/vertex level are polygons that have intersecting bounding volumes. This will typically be a few polygons at most for each polygon being tested. That is certainly a lot more efficient than testing every single vertex and every single polygon against every edge of every other polygon in the scene.

RepairTJunctions - CBaseTree

The code to this function is quite small and straightforward. It is passed two polygon pointers. The first polygon is the polygon currently having its edges tested for T-junctions. It is the polygon that will have vertices added to it if a T-junction is identified. The second polygon is the polygon that is going to have each of its vertices tested against each of the edges of the first polygon to see if any of them share that edge and cause a T-junction.

The function begins by setting up a loop to iterate through every edge in the polygon. The loop will step through each vertex and the vertex ahead of it in the array for the edge that is currently being tested.

```
void CBaseTree::RepairTJunctions( CPolygon * pPoly1, CPolygon * pPoly2 )
```

```
{
    D3DXVECTOR3 Delta;
    float       Percent;
    ULONG       v1, v2, v1a;
    CVertex     Vert1, Vert2, Vert1a;

    // Validate Parameters
    if (!pPoly1 || !pPoly2) return;

    // For each edge of this face
    for ( v1 = 0; v1 < pPoly1->m_nVertexCount; v1++ )
    {
        // Retrieve the next edge vertex (wraps to 0)
        v2 = ((v1 + 1) % pPoly1->m_nVertexCount);

        // Store verts (Required because indices may change)
        Vert1 = pPoly1->m_pVertex[v1];
        Vert2 = pPoly1->m_pVertex[v2];
```

v1 is always the index of the current vertex and v2 is the index of the next one in the winding of the polygon. These two vertex indices form the current edge that is to be tested against. Notice that v2 is calculated using the modulus operator so that when v1 is equal to the last vertex in the winding, v2 will wrap around to vertex 0 (the final edge in a polygon is formed by vertex N and vertex 0, where N is the number of vertices in the polygon). In the above code you should notice that once these indices are calculated, we use them to fetch the actual vertex data from the CPolygon's vertex array.

We now have the two vertices of the edge we wish to test, so now we need to loop through each vertex in the second polygon and calculate its distance from the edge. If the DistanceToLineSegment method that we just wrote is passed the two vertices of the edge and the vertex from polygon 2, and it returns zero (with tolerance), the vertex of the second polygon causes a T-junction with the edge and steps must be taken to repair it.

```
        // Now loop through each vertex in the test face
        for ( v1a = 0; v1a < pPoly2->m_nVertexCount; v1a++ )
        {
            // Store test point for easy access
            Vert1a = pPoly2->m_pVertex[v1a];

            // Test if this vertex is close to the test edge
            if ( MathUtility::DistanceToLineSegment(  (D3DXVECTOR3&)Vert1a,
                                                      (D3DXVECTOR3&)Vert1,
                                                      (D3DXVECTOR3&)Vert2 )<0.01f )
            {
```

In this next code block we repair the T-junction. This is done by simply inserting a new vertex in the polygon immediately after the first vertex in the edge we are testing (v1). That is, we must insert it at the position in the vertex array currently occupied by the second vertex v2. This will nudge all vertices in the winding order up by one position in the array, creating the current edge between v1 and the new vertex we have just inserted. It also adds a new edge to the polygon between the new vertex and what used to be the second vertex in the current edge.

The following code demonstrates inserting a new vertex into the polygon and setting its position equal to the vertex in the second polygon that caused the T-junction.

```
                // Insert a new vertex within this edge
                long NewVert = pPoly1->InsertVertex( (USHORT)v2 );
                if (NewVert < 0) throw std::bad_alloc();

                // Set the vertex pos
                CVertex * pNewVert = &pPoly1->m_pVertex[ NewVert ];

                pNewVert->x = Vert1a.x;
                pNewVert->y = Vert1a.y;
                pNewVert->z = Vert1a.z;
```

Because our polygon has a vertex in exactly the same position as the one that caused the T-junction in the second polygon, we have repaired it. However, we also have to generate the texture coordinate information and normal for this newly inserted vertex.

Just as we did in the polygon splitting routine, we create the new normal and texture coordinates for the vertex by interpolating the texture coordinates and normal vectors stored in the original edge vertices based on the distance along that edge where we have inserted the new vertex. In the following code we divide the length of the new edge (vert1 to new vertex) by the length of the original edge (Vert1 to Vert2) to get a *t* value back that describes the position of the new vertex parametrically along the original edge. We use this to weight the interpolation as we have done so many times before. Below we see the remainder of the function.

```
                // Calculate the percentage for interpolation calcs
                Percent = D3DXVec3Length( &(*(D3DXVECTOR3*)pNewVert -
                                            (D3DXVECTOR3&)Vert1) )
                          /
                          D3DXVec3Length( &((D3DXVECTOR3&)Vert2 -
                                           (D3DXVECTOR3&)Vert1) );

                // Interpolate texture coordinates
                Delta.x      = Vert2.tu - Vert1.tu;
                Delta.y      = Vert2.tv - Vert1.tv;
                pNewVert->tu = Vert1.tu + ( Delta.x * Percent );
                pNewVert->tv = Vert1.tv + ( Delta.y * Percent );

                // Interpolate normal
                Delta            = Vert2.Normal - Vert1.Normal;
                pNewVert->Normal = Vert1.Normal + (Delta * Percent);
                D3DXVec3Normalize( &pNewVert->Normal, &pNewVert->Normal );

                // Update the edge for which we are testing
                Vert2 = *pNewVert;

            } // End if on edge
        } // Next Vertex v1a
    } // Next Vertex  v1
}
```

Not only have we learned how to fix T-junctions, we have also finished our coverage of CBaseTree for this chapter. In the next chapter we will add a few more functions and variables to CBaseTree so that it offers an efficient rendering strategy. For now though, we will remain focused on tree building and using trees to perform collision queries.

With the base functionality in place, we can now finally look at the code to the derived classes. For the most part, these classes will contain only a few methods. This is due to the fact that, apart from the building and traversal functions, everything else has already been implemented in CBaseTree.

In the following sections of the textbook we will cover the source code to the quad-tree, the oct-tree and the kD-tree, one at a time and in that order. Most of their implementations will be identical, with only slightly different traversal and build logic. Thus, covering the building process for each tree should be a relatively painless affair after we have covered the build function for the first. Let us start by looking at our first derived class, CQuadTree. This is the class that implements the vanilla quad-tree and it is the first tree class we have developed so far that is designed to be instantiated by the application.

14.17 CQuadTree - Implementation

Contained in the source files CQuadTree.cpp and CQuadTree.h is the code for our vanilla quad-tree implementation. It is derived from CBaseTree as all tree classes will be and as such, there are several functions that we must implement because they are required by ISpatialTree but not implemented in CBaseTree. For example, we must implement the Build method. This method will recursively build a tree of quad-tree nodes using the polygon and detail area data that has been registered with the tree and populate the leaf lists with CBaseLeaf objects. After the Build function has returned, the quad-tree will be complete and we will have a tree of quad-tree nodes. Each node will contain a bounding box large enough to contain the polygon data and detail areas that are assigned to any of its child nodes. At each point during the build process, a node will be divided into four equal quadrants and each quadrant assigned to a child of that node.

Other functions that are specified in ISpatialTree but not implemented in CBaseTree are the collision functions, such as CollectLeavesAABB and CollectLeavesRay. These must also be implemented in this class to provide the traversal logic that steps through the nodes of a quad-tree searching for leaf nodes. These will be small and simple methods since the collision code and much of the core logic resides in CBaseTree and CCollision.

14.17.1 CQuadTree Node – The Source Code

Although CBaseTree implements the CBaseLeaf object that all trees will use to contain the polygon and detail area data at a leaf node, it does not implement a node object that will be used to link the nodes of the hierarchy together. This is because the node structure of each derived class will be different since they have different child counts. A quad-tree for example must store pointers to four child nodes, an oct-tree must store pointers to eight children, and a kD-tree will store only two child node pointers. This

means it is up to the programmer who implements the derived class to also define the node structure the tree will use. The structure we use for the quad-tree nodes is shown 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-to-frame coherence plane
};
```

This very simple structure contains pointers to four children and a CBaseLeaf pointer. The only nodes that will not have NULL assigned to their leaf pointer will be terminal nodes (nodes at the ends of branches that have been made into leaf nodes). When traversing the tree searching for leaf nodes, we know that a leaf node is any node that does not have NULL assigned to its leaf pointer. If it is leaf node, the attached CBaseLeaf object will contain all the polygon data and detail area data assigned to that terminal node. Following the leaf pointer in the declaration are two vectors in which the node's bounding box will be stored and a fifth member that will be explained in detail in the next lesson.

Finally, notice that this class has a method called SetVisible which the tree can call to make all its children visible. A node does not hold a visible status but this function is a recursive function that traverses the children of the node and sets the visibility status of any leaves underneath it to true. This will be used in our visibility testing methods. For example, if a node's volume is completely contained inside the frustum, then all direct and indirect children of that node must also be inside it. When this is the case, we no longer have to perform frustum/AABB tests on each of its children since we know the entire branch starting at that node must be contained in the frustum. When this is the case, this method is called to immediately set the visibility status of all child leaves to true.

Constructor - CQuadTreeNode

The quad-tree node constructor makes sure that its leaf and child pointers are set to NULL and that the last frustum plane index is set to -1 (no plane).

```
CQuadTreeNode::CQuadTreeNode()
{
    unsigned long i;

    // Reset / Clear all required values
```

```
    Leaf             = NULL;
    LastFrustumPlane = -1;

    // Delete children
    for ( i = 0; i < 4; ++i ) Children[i] = NULL;
}
```

Destructor - CQuadTreeNode

The destructors for all of our derived node types will work the same way. When a node is destroyed, it deletes its child nodes, which causes their destructors to trigger, which then deletes their child nodes, and so on down the tree. This causes a cascade effect that removes every node of the tree from memory. Thus, the tree just has to delete the root node to kick off this process. The code is shown below.

```
CQuadTreeNode::~CQuadTreeNode()
{
    unsigned long i;

    // Delete children
    for ( i = 0; i < 4; ++i )
    {
        if ( Children[i] ) delete Children[i];
        Children[i] = NULL;

    } // Next Child

    // Note : We don't own the leaf, the tree does, so we don't delete it here

    // Reset / Clear all required values
    Leaf = NULL;
}
```

Note that if the node is a terminal node and has a leaf attached to it, we do not delete the leaf from memory, we simply set its pointer to NULL. The reason we do not worry about the node deleting its leaf is because the leaf pointers are also stored in the tree's m_Leaves STL list. The tree can delete all the leaves in one go afterwards simply by emptying this list.

Since the constructor and destructor for all of our various node types will be the same as this one (just with more or less children to delete) we will not show the code to the constructors and destructors of other node types from this point forwards.

SetVisible - CQuadTreeNode

Although we will not see the SetVisible method being used until we cover the rendering system of CBaseTree (in the next lesson), its code is very simple so we will cover it here. As you can see, it first tests to see if its leaf pointer is non NULL. If it is, then this is a leaf and we set the leaf's visible status to that of the boolean passed. This means this recursive function can be used to turn on or off the visibility

status of all leaves down a given branch of the tree (using the boolean parameter). If the current node is a leaf node then our job is done and we return. If the current node is not a leaf, then the function calls itself recursively for each of its children.

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

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

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

    } // Next Child
}
```

14.17.2 CQuadTree – The Source Code

Below we see the CQuadTree class declaration (see CQuadTree.h) which shows the functions and members we added beyond those inherited from CBaseTree. In the following listing you can see that the first group of methods are those that must be implemented to support the ISpatialTree interface. For example, we know that every derived class must provide the implementation for the Build, CollectLeavesAABB, and the CollectLeavesRay methods since these are the application's only means to build and query the tree. Even methods in CBaseTree, such as the Repair method, rely on the CollectLeavesAABB method being implemented in the derived class so that polygon neighbors can be quickly determined.

Let us look at the public methods implemented to the support the base class first.

```
class CQuadTree : public CBaseTree
{
public:

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

    // Public Virtual Functions for This Class (from base).
    virtual bool Build                  ( bool bAllowSplits = true );
    virtual void ProcessVisibility      ( CCamera & Camera );

    virtual bool CollectLeavesAABB      ( LeafList & List,
                                          const D3DXVECTOR3 & Min,
                                          const D3DXVECTOR3 & Max );

    virtual bool CollectLeavesRay       ( LeafList & List,
                                          const D3DXVECTOR3 & RayOrigin,
```

```
                                        const D3DXVECTOR3 & Velocity );
    virtual void DebugDraw              ( CCamera & Camera );

    virtual bool GetSceneBounds         ( D3DXVECTOR3 & Min, D3DXVECTOR3 & Max );
```

The ProcessVisibility method is specified in ISpatialTree and should be called by the application prior to calling the ISpatialTree::DrawSubset method. Although we will not discuss the rendering system until the next chapter, the purpose of this function is to traverse the quad-tree with the passed camera and set the visible status of any leaves according to whether they are currently inside or outside the frustum. The GetSceneBounds function should return the bounding box of the entire area compiled into the quad-tree (which is the bounding volume of the root node). Finally, the DebugDraw method does not have to be implemented in any derived class as it has a default implementation in ISpatialTree which does nothing. However, each of our derived classes will override this method so that we can traverse the tree and render their leaf node bounding boxes to visualize what we have created.

Below we see the private class methods. These are basically the methods that the public methods shown above use to accomplish their tasks. The functions listed above are really just doorways to the private recursive procedures. For example, the Build method creates the root node and builds the initial polygon list for that node and then calls the BuildTree function which recursively calls itself until the tree is complete. The same is true of the CollectLeavesAABB method which is really a one line function that makes sure the CollectAABBRecurse method is first called using the root node.

```
private:

    // Private Functions for This Class
    bool         BuildTree              ( CQuadTreeNode * pNode,
                                          PolygonList PolyList,
                                          DetailAreaList AreaList,
                                          const D3DXVECTOR3 & BoundsMin,
                                          const D3DXVECTOR3 & BoundsMax );

    void         UpdateTreeVisibility   ( CQuadTreeNode * pNode,
                                          CCamera & Camera,
                                          UCHAR FrustumBits = 0x0 );

    bool         DebugDrawRecurse       ( CQuadTreeNode * pNode,
                                          CCamera & Camera,
                                          bool bRenderInLeaf );

    bool         CollectAABBRecurse     ( CQuadTreeNode * pNode,
                                          LeafList & List,
                                          const D3DXVECTOR3 & Min,
                                          const D3DXVECTOR3 & Max
                                                bool            bAutoCollect = false
);

    bool         CollectRayRecurse      ( CQuadTreeNode * pNode,
                                          LeafList & List,
                                          const D3DXVECTOR3 & RayOrigin,
                                          const D3DXVECTOR3 & Velocity );
```

Because CBaseTree provides us with the leaf list, detail area list, and polygon list in which to store our data, we only need to add a few more members in this class. The first is a CQuadTreeNode pointer which will point to the root node of the tree once it has been built. Given our extensive exposure to hierarchies in this course, we know that we only need to store the root node to a tree/hierarchy for its methods to be able to perform a complete traversal of every node in the tree. We also add a boolean member called m_bAllowSplits which simply stores whether a clipped tree is being built. This is determined by the value of the boolean parameter passed into the build function, which is stored in this member. The remainder of the class declaration is shown below.

```
      //-----------------------------------------------------------------------
      // Private Variables for This Class
      //-----------------------------------------------------------------------
      CQuadTreeNode * m_pRootNode;      // The root node of the tree
      bool            m_bAllowSplits;   // Is splitting allowed?

      // Stop Codes
      float          m_fMinLeafSize;    // Min leaf size stop code
      ULONG          m_nMinPolyCount;   // Min polygon count stop code
      ULONG          m_nMinAreaCount;   // Min detail area count stop code
};
```

Notice that we now have three members at the very bottom which store a minimum leaf size, a minimum polygon count, and a minimum detail area count. The values of these members will be passed in as parameters to the constructor by the application to provide a degree of control over the way the tree is built. When the tree is being compiled, a decision must be made at each node as to whether the node satisfies the requirements to become a leaf node. We implement three stop codes to that can be set by the application to influence this decision. If any of these stop codes are reached, the current node will have no children generated for it and will be made a leaf node. The three stop codes we use for the quad-tree are defined in CQuadTree.h and are discussed next.

The first stop code is the size of the leaf. We currently set our default such that, if the bounding volume of a node is determined to have a diagonal length (e.g., bottom left to top right) of less than 300 world space units, we decide that this node's volume is so small that we do not wish to further subdivide and create any more children down that branch of the tree. When this is the case, the node's child pointers are set to NULL and a new CBaseLeaf object is created and attached to the node. The leaf is also added to the tree's leaf list and the leaf itself has the pointers for all polygons and detail areas that made it into that node added to its internal arrays.

The second stop code occurs during the creation of a node when we determine that less than a specified number of polygons have made it into that node. The default value for this is 600. This means that regardless of how large a node's volume might be, if it contains less than 600 polygons we decide that further subdividing this polygon set would be futile and the node is made a leaf, exactly as described in the previous paragraph.

The third stop code is one that allows us to stop subdividing a node's volume if less than a certain number of detail areas exist in that node. By default we set this to zero so that detail areas play no part in determining whether a node should be made a leaf node. However, if you were to set this value to 1 (for example) then whenever a node was encountered that had only one detail area in it and the polygon list was lower than the polygon count stop code, a leaf node will be created. This stop code may be a little

hard to understand, and most of the time it will go unused and be set to zero. However, it can become very useful when you are building a tree that has no polygon data and consists only of detail areas, since it allows us to control the size of the leaf nodes under those conditions. For example, consider what happens if we compiled our terrain into a quad-tree. As discussed previously in this lesson, we will not want to compile every terrain polygon into the tree since the terrain data is already organized into a series of render ready terrain blocks (sub-meshes). What we might decide to do instead is just build a tree of detail areas. We could for example create a bounding box around each terrain block and register each of these volumes as detail areas with the tree. When the tree is compiled (which contains no polygon data) the only stop code would be the leaf size. However, this might carve up the space into many more leaves than necessary such that a given terrain block's volume could be contained in many leaves. If all we are using the tree for is quickly querying which terrain block is visible, we would rather have each terrain block fit into a single leaf equal to its size. Ideally, if the terrain was divided into a 10x10 grid of terrain blocks, we would like that space divided up into a grid of 10x10 terrain block sized leaf nodes. In this situation we could set the detail area count stop code to 1, and as soon as a node had only one detail area assigned to it, it would be made into a leaf node containing that single detail object. This means our tree would contain a leaf node for each terrain block allowing us to keep the tree as shallow as possible and therefore speeding up tree traversals.

The Constructor - CQuadTree

The CQuadTree constructor accepts two required parameters and three optional parameters. The first is a pointer to a Direct3D device which the rendering system will use to render the tree. The second is a boolean describing whether this is a hardware or software device. This information will be needed by the rendering system when creating the index and vertex buffers that will store the geometry for the render system.

```
CQuadTree::CQuadTree( LPDIRECT3DDEVICE9 pDevice,
                      bool bHardwareTnL
                      float fMinLeafSize /* = 300.0f */,
                      ULONG nMinPolyCount /* = 600 */,
                      ULONG nMinAreaCount /* = 0 */ ) : CBaseTree( pDevice, bHardwareTnL )
{
    // Clear required variables
    m_pRootNode = NULL;

    // Store stop code values
    m_fMinLeafSize  = fMinLeafSize;
    m_nMinPolyCount = nMinPolyCount;
    m_nMinAreaCount = nMinAreaCount;
}
```

The device pointer and its boolean hardware status are immediately sent to the CBaseTree constructor where they are stored in CBaseTree member variables. These will be used after the tree is built to set up the render system. The final three parameters contain the stop codes that the application can pass to influence which nodes are candidates for becoming leaf nodes during a build. We simply store these values in their corresponding member variables.

In our coverage of CBaseTree we learned that the CBaseTree::PostBuild method was to be called after the derived class's Build method completed its task and built the tree (CBaseTree::PostBuild will be called from the CQuadTree::Build function just before it returns). The PostBuild method issues a call to the CBaseTree::BuildRenderData method which we will cover in the next lesson. This method will build the render data if NULL was not passed as the device pointer. Therefore, we can inform the tree that it will not be used for rendering by passing NULL as the first parameter and save the memory that would otherwise be allocated to prepare and contain the tree data in a renderable format.

Destructor – CQuadTree

At first glance, the CQuadTree destructor appears to neglect to free a lot of its memory. It simply deletes the root node, which triggers the deletion of every node in the tree.

```
CQuadTree::~CQuadTree()
{
    // Release any resources
    if ( m_pRootNode ) delete m_pRootNode;

    // Clear required variables
    m_pRootNode = NULL;
}
```

What about the leaves, the polygons and the detail area? Keep in mind that our classes all have virtual destructors, so the base class's destructor will automatically be called. CBaseClass actually manages the polygon, detail area, and leaf list data, so it is this destructor that releases them (shown below).

As you can see, the base class destructor first loops through the linked list of polygons used by the tree and deletes each one before emptying the pointer list and freeing up its memory.

```
CBaseTree::~CBaseTree()
{
    PolygonList::iterator       PolyIterator = m_Polygons.begin();
    LeafList::iterator          LeafIterator = m_Leaves.begin();
    DetailAreaList::iterator    AreaIterator = m_DetailAreas.begin();
    ULONG                       i;

    // Iterate through any stored polygons and destroy them
    for ( ; PolyIterator != m_Polygons.end(); ++PolyIterator )
    {
        CPolygon * pPoly = *PolyIterator;
        if ( pPoly ) delete pPoly;

    } // Next Polygon

    m_Polygons.clear();
```

We then iterate through every detail area in the tree's detail area STL list, delete each one and empty that list too.

```
    // Iterate through any stored detail area objects and release
    for ( ; AreaIterator != m_DetailAreas.end(); ++AreaIterator )
    {
        TreeDetailArea * pDetailArea = *AreaIterator;
        if ( pDetailArea ) delete pDetailArea;

    } // Next detail area
    m_DetailAreas.clear();
```

And of course, we do the same for the list of leaves.

```
    // Iterate through the leaves and destroy them
    for ( ; LeafIterator != m_Leaves.end(); ++LeafIterator )
    {
        CBaseLeaf * pLeaf = (CBaseLeaf*)(*LeafIterator);
        if ( pLeaf ) delete pLeaf;

    } // Next Leaf

    m_Leaves.clear();
```

We then release the 3D device that the base tree render system will use. This device was passed in and stored in the constructor. We will see it being used in the following lesson.

```
    // Release other D3D Objects
    if ( m_pD3DDevice ) m_pD3DDevice->Release();

    // Clear required variables
    m_pD3DDevice        = NULL;
}
```

Just remember later when examining the destructors of each derived class, that this base class constructor will always be called too. This is how the main memory for much of the tree gets released. All the derived class has to do in its destructor is delete the root node.

Build - CQuadTree

We finally find ourselves at the point where we can discuss the code that actually builds a tree. The application should call the CQuadTree::Build function only after it has registered all static polygon and detail area data with the tree (using the AddPolygon and AddDetailArea methods inherited from the base class). It is important that at the time this function is called, the object's m_Polygons STL list contains all of the static polygon data we wish to compile into the tree and the m_DetailAreas array contains all of the detail areas that we would like compiled into the tree.

This function is not the recursive function that calls itself until the tree is fully built, but it is the function that allocates the root node and prepares the scene data for the recursive process (invoked by a call to the CQuadTree::BuildTree method). This function has the following tasks to perform:

1. Allocate a new CQuadTreeNode for the root node of the tree.

2. Loop through each polygon in its list and compile an AABB for the data.

3. Loop through each detail area and possibly adjust the AABB computed in step 2 so that all polygon data and detail areas are contained inside the bounding box (this will be the bounding box of the root node).

4. Build a list of all the polygons contained in this node (which will actually be a new temporary list containing all the polygons in the entire scene). Since this is the root node, this list will initially be a copy of the complete m_Polygons list (more on this in a moment). It is this list that will be passed down the tree and divided into four child lists at each node.

5. Compile a list of all the detail area in the root node. This list will initially be an exact copy of the m_DetailAreas list to begin with since this is the root node. As this list is passed down the tree, it will be divided into four lists at each node, one for each child.

6. Call the recursive BuildTree function passing the node pointer (step 1), the list of polygons and detail areas (steps 4 and 5) and the bounding box (steps 2 and 3). The recursive function will then store that passed AABB in the node and will sort the passed polygon and detail area lists into four lists, one for each child it creates. Then it will call itself recursively for each child, passing in the child node and the polygon and detail area lists for that node.

7. When the recursive function returns, call the CBaseTree::PostBuild method. This will instruct the base class to calculate the polygon bounding boxes and initialize the render system. If a valid device pointer was passed to the constructor, the initialization of the render system will be triggered which will also issue a call to the CBaseTree::Repair method to repair all T-junctions that exist in the tree's polygon set.

Before we look at the code, we have one more small matter to address. If a clipped tree is not being compiled, then after the build is finished, the m_Polygons list should and will contain the same polygon pointers that were originally registered with the tree (i.e., the building process will not alter the polygons stored in this list). The CPolygon pointers stored in this list will be passed down the tree and a copy of each polygon pointer will be stored in the leaves in which it is contained. As the polygon data is never clipped or altered, the m_Polygons array will not be altered by the build process and will always contain the polygons contained in the tree even after the build process.

However, if we pass true as the (only) parameter into the Build function, we are informing the tree that we would like the polygon data clipped to the leaf nodes so that no polygon will ever be stored in more than one leaf. We know that when we clip a polygon, we get back two new polygons and the original polygon is deleted and replaced by the two new clips. After the build process has completed in the case of a clipped tree, we do not want the m_Polygons array to contain the same polygons that were registered with the scene as these will no longer describe the polygons being used by the tree. Several polygons in that list will have been deleted the moment they were split, and many new polygons would have been added.

Fortunately there is not much to worry about. As we discussed in the steps above, we copy the polygon pointers into a temporary polygon list for the root node anyway, and it is this list that is passed down the tree. Therefore, in the clipped case, once we have made that temporary copy of the list, we will empty the m_Polygons list so that it contains no data. During the build process, polygons in the temporary list that gets passed to each node will be deleted and split into two new polygons which will all be added to these temporary lists. However, as soon as we find a leaf node, we will be passed a list of clipped polygons that fit exactly inside that leaf. We will then add these pointers to the m_Polygons list as and when they make it into a leaf. That way, as each leaf is created, its new split polygons are added back into the m_Polygons list, leaving us with an m_Polygons list at the end of the build process which contains all the polygons actually being used by the tree. This will be a very different list versus the original polygon set prior to the build process.

Let us now look at the code a section at a time.

The function is passed a single parameter which determines whether a clipped or non-clipped tree should be built. If this boolean is set to false (the default) a non-clipped tree will be built. Take a look at the variables we declare on the stack at the head of the function.

```
bool CQuadTree::Build( bool bAllowSplits /* = false */ )
{
    PolygonList::iterator       PolyIterator = m_Polygons.begin();
    DetailAreaList::iterator    AreaIterator = m_DetailAreas.begin();
    PolygonList                 PolyList;
    DetailAreaList              AreaList;
    unsigned long               i;

        // Reset our tree info values.
    D3DXVECTOR3 vecBoundsMin( FLT_MAX, FLT_MAX, FLT_MAX );
    D3DXVECTOR3 vecBoundsMax( -FLT_MAX, -FLT_MAX, -FLT_MAX );
```

We declare two iterators that will be used to step through the m_Polygons and m_DetailAreas lists. These lists contain the data we wish to compile. However, we also instantiate a local scope polygon list and detail area list that will be used to contain copies of the m_Polygons and m_DetailAreas lists respectively. We then set the vecBoundsMax and vecBoundsMin vectors which will be used to record the size of the root node's volume (an inside out box initially).

Next we allocate the root node of the tree and store its pointer in the CQuadTree member variable. We also copy the value of the bAllowSplits boolean into the m_bAllowSplits member variable.

```
    // Allocate a new root node
    m_pRootNode = new CQuadTreeNode;
    if ( !m_pRootNode ) return false;

    // Store the allow splits value for later retrieval.
    m_bAllowSplits = bAllowSplits;
```

Our next task is to loop through every polygon in the m_Polygons list and adjust the extents of the bounding box such that it is large enough to contain the vertices of all polygons in the scene. At the end of the following loop, vecBoundsMin and vecBoundsMax will represent a box that bounds all the

polygon data in the root node. Note that as we process each CPolygon pointer in m_Polygons and adjust the box to fit it, we add its pointer to the local polygon list declared at the top of the function. Therefore, at the end of this loop, we will have also made a copy of all the polygon pointers contained in m_Polygons into the local PolyList.

```
    // Loop through all of the initial polygons
    for ( ; PolyIterator != m_Polygons.end(); ++PolyIterator )
    {
        // Retrieve the polygon
        CPolygon * pPoly = *PolyIterator;
        if ( !pPoly ) continue;

        // Calculate total scene bounding box.
        for ( i = 0; i < pPoly->m_nVertexCount; ++i )
        {
            // Store info
            CVertex * pVertex = &pPoly->m_pVertex[i];
            if ( pVertex->x < vecBoundsMin.x ) vecBoundsMin.x = pVertex->x;
            if ( pVertex->y < vecBoundsMin.y ) vecBoundsMin.y = pVertex->y;
            if ( pVertex->z < vecBoundsMin.z ) vecBoundsMin.z = pVertex->z;
            if ( pVertex->x > vecBoundsMax.x ) vecBoundsMax.x = pVertex->x;
            if ( pVertex->y > vecBoundsMax.y ) vecBoundsMax.y = pVertex->y;
            if ( pVertex->z > vecBoundsMax.z ) vecBoundsMax.z = pVertex->z;

        } // Next Vertex

        // Store this polygon in the top polygon list
        PolyList.push_back( pPoly );

    } // Next Polygon

    // Clear the initial polygon list if we are going to split the polygons
    // as this will eventually become storage for whatever gets built
    if ( bAllowSplits ) m_Polygons.clear();
```

Notice what we do in the very last line of code show above. If a clipped tree is being built, we empty m_Polygons so that it no longer contains any polygon data. We can do this because we now have a copy of each pointer in the local list. m_Polygons will be repopulated with the clipped polygon data as each leaf node is encountered and created. If a clipped tree is not being built, we do not empty this list since it will be unchanged during the build process.

We now have a bounding box for the root node polygon data but we must also make sure that it is large enough to also bound any registered detail areas. So we now loop through each registered detail area and adjust the root node's bounding box to contain the bounding boxes of all detail areas. As with the above loop, as we process each detail area, we also copy its pointer into the local detail area list (DetailList) so that we have a complete list of detail areas which can be passed into the recursive process.

```
    // Loop through all of the detail areas
    for ( ; AreaIterator != m_DetailAreas.end(); ++AreaIterator )
    {
        // Retrieve the detail area
        TreeDetailArea * pDetailArea = *AreaIterator;
```

```
        if ( !pDetailArea ) continue;

        // Calculate total scene bounding box.
        D3DXVECTOR3 & Min = pDetailArea->BoundsMin;
        D3DXVECTOR3 & Max = pDetailArea->BoundsMax;

        if ( Min.x < vecBoundsMin.x ) vecBoundsMin.x = Min.x;
        if ( Min.y < vecBoundsMin.y ) vecBoundsMin.y = Min.y;
        if ( Min.z < vecBoundsMin.z ) vecBoundsMin.z = Min.z;
        if ( Max.x > vecBoundsMax.x ) vecBoundsMax.x = Max.x;
        if ( Max.y > vecBoundsMax.y ) vecBoundsMax.y = Max.y;
        if ( Max.z > vecBoundsMax.z ) vecBoundsMax.z = Max.z;

        // Store this in the top detail area list
        AreaList.push_back( pDetailArea );

    } // Next Polygon
```

At this point we have a root node pointer, a bounding box large enough to contain all the data contained in that node, and a complete list of all the registered polygons and detail areas stored in the PolyList and AreaList local lists. We are now ready to start the internal build process, so we pass this information into the BuildTree recursive function.

```
    // Build the tree itself
    if ( !BuildTree(m_pRootNode, PolyList, AreaList, vecBoundsMin, vecBoundsMax ))
         return false;
```

This function call will store the passed bounding box in the passed node, which means in its first iteration it will store the bounding box we have just compiled in the root node. It will then divide this bounding box into four quadrants creating the bounding boxes for each of the four child nodes. Then it will classify the polygon list and the detail area list against each bounding box, creating four separate polygon and detail area lists for each child. The four child nodes are then created and the BuildTree function calls itself recursively for each child and the process repeats until the function determines it has been passed a node that should be made into a leaf. This function will be covered in a moment, but for now just know that when it returns program flow back to CQuadTree::Build, the entire hierarchy of nodes will have been created and m_Polygons will contain the polygon data being used by the tree.

At this point, the tree is fully constructed. Before we return program flow back to the application, we issue a call to the CBaseTree::PostBuild function which instructs the base class to calculate the bounding boxes of each polygon and prepare the data for rendering (which includes T-junction repair).

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

The recursive CQuadTree::BuildTree function is true core of this system, so let us cover that next.

BuildTree – CQuadTree

BuildTree is the main tree compilation function. It is passed a node, a list of polygons and detail areas that fit in the bounding volume of that node, and the extent vectors of its volume. The function will first assign the passed volume to the passed node. Thus, the first time this function is called, BoundsMin and BoundsMax will be assigned to the root node as its bounding volume. It also means that the first time it is called, PolyList and AreaList will contain the entire scene since the root node's volume bounds everything (assuming you are using the tree for an entire scene of course). In the case of the root node, its bounding volume was calculated in the previously discussed function and passed in along with the node. For all other nodes however, this function will generate the bounding boxes and polygon lists for each of its child nodes and call itself recursively. Therefore, the second time this function is called it will be called by the previous instance of the function. The node pointer passed will not be the root, it will be the first child of the root. The PolyList and AreaList will contain only the geometry that was contained in the root node's first child, and BoundsMax and BoundsMin will be the bounding volume of that child node quadrant.

Let us first look at the variables that are allocated on the stack. We will need to be able to iterate through the passed polygon and detail area lists so that we can divide them into four sub-lists (one per child), so we instantiate two iterators for this purpose. We will also need four temporary lists to hold the polygon data for each child node and the detail areas for each child node, so you can see that we instantiate an array of four PolygonLists (an STL list of CPolygons) and four DetailAreaLists (an STL list of detail area structures). We allocate two D3DXPLANE structures that will be used to create and store the two split planes at this node that divide the node into four quadrants. We will see these being used in a moment.

Again, the first thing we do in the following code is assign the passed volume to the passed node.

```
bool CQuadTree::BuildTree( CQuadTreeNode * pNode,
                           PolygonList PolyList,
                           DetailAreaList AreaList,
                           const D3DXVECTOR3 & BoundsMin,
                           const D3DXVECTOR3 & BoundsMax )
{
    D3DXVECTOR3                 Normal;
    PolygonList::iterator       PolyIterator;
    DetailAreaList::iterator    AreaIterator;
    CPolygon                  * CurrentPoly, * FrontSplit, * BackSplit;
    PolygonList                 ChildList[4];
    DetailAreaList              ChildAreaList[4];
    CCollision::CLASSIFYTYPE    Location[2];
    D3DXPLANE                   Planes[2];
    unsigned long               i;
    bool                        bStopCode;

    // Store the bounding box properties in the node
    pNode->BoundsMin = BoundsMin;
    pNode->BoundsMax = BoundsMax;
```

Before we start subdividing space we need to determine whether or not the leaf is so small or its polygon list contains so few polygons that this node should be made a terminal node (a leaf). We discussed earlier that there are three stop codes that if true, prevent us creating any more children. We simply store the passed polygon data in the node, making it a leaf. Therefore, we first calculate the diagonal length of the current node's bounding box and test this against the m_fMinLeafSize stop code.

```
    // Calculate 'Stop' code
    D3DXVECTOR3 vecLeafSize = D3DXVECTOR3( BoundsMax.x - BoundsMin.x,
                                           0,
                                           BoundsMax.z - BoundsMin.z );

    bStopCode  = (AreaList.size() == 0 && PolyList.size() == 0) ||
                 (AreaList.size() <= m_nMinAreaCount  &&
                  PolyList.size()<= m_nMinPolyCount ) ||
                  D3DXVec3Length( &vecLeafSize ) <= m_fMinLeafSize;
```

Let us analyze the multiple conditionals being used above. First, if there are no detail areas in the passed area list and no polygons in the passed polygon list, the node is totally empty and the boolean bStopCode will be set to true. In that case, we will create a leaf here. This boolean is also set to true if the number of detail areas is less then or equal to the m_nMinAreaCount stop code or if the number of polygons is less than or equal to the minimum polygon count amount. So, if we have too few polygons and too few detail areas in this node (as defined by the stop codes), we set bStopCode to true since we wish to halt subdivision and make this node a leaf. The final test that sets bStopCode to true happens when the length of the diagonal vector from the node's extents is less than or equal to the minimum leaf size stop code. Notice that because we are using a quad-tree, all nodes will have the same height and we do not factor in the y components of the extents (we set the vecLeafSize vector's Y component to zero). For a quad-tree, the diagonal vector projected on to the XZ plane is all we are concerned with.

At this point we have a boolean that tells us whether we should continue to divide this node into four children or whether we should just make a new leaf here and return.

In the next section of code we see what happens if this node is to become a leaf. We first allocate a new CBaseLeaf structure, passing in a pointer to the current tree instance, and then loop through every polygon in the passed list and add it to the leaf's polygon array using the CBaseLeaf::AddPolygon function. At the end of this loop all the polygons that made it into the node will be stored in a new CBaseLeaf.

```
    // If we reached our stop limit, build a leaf here
    if ( bStopCode )
    {
        // Build a leaf
        CBaseLeaf * pLeaf = new CBaseLeaf( this );
        if ( !pLeaf ) return false;

        // Store the polygons
        for ( PolyIterator = PolyList.begin();
              PolyIterator != PolyList.end();
              ++PolyIterator )
        {
            // Retrieve poly
```

```
            CurrentPoly = *PolyIterator;
            if ( !CurrentPoly ) continue;

            // Add to full tree polygon list ONLY if splitting is allowed
            if ( m_bAllowSplits ) AddPolygon( CurrentPoly );

            // Also add a reference to the leaf's list
            pLeaf->AddPolygon( CurrentPoly );

        } // Next Polygon
```

Notice something very important in the above code. If a clipped tree is being built, we need to repopulate the tree's m_Polygons array which was emptied at the start of the build process. We do this using the CBaseTree::AddPolygon function. This way, when all leaves have been created for the tree, the m_Polygons list will be fully populated with all the polygon fragments that made it into each clipped leaf. Obviously, we do not add the polygons to the tree's list if we are not building a clipped tree as they are already stored there.

Our next task is to loop through each detail area that made it into this node's list and add each of those to the new leaf's detail area array. The CBaseLeaf::AddDetailArea method which we covered earlier will be used to perform this task.

```
        // Store the area lists
        for ( AreaIterator = AreaList.begin();
              AreaIterator != AreaList.end(); ++AreaIterator )
        {
            // Retrieve detail area item
            TreeDetailArea * pDetailArea = *AreaIterator;
            if ( !pDetailArea ) continue;

            // Add a reference to the leaf's list
            pLeaf->AddDetailArea( pDetailArea );

        } // Next Polygon
```

At this point we have a new leaf and it contains (in its internal arrays) a list of all the CPolygon pointers and detail area pointers that made it into this node. We will now assign the node's Leaf pointer to point at our new leaf object and set the bounding box of the leaf to be the same as the node's bounding box (using the CBaseLeaf::SetBoundingBox method to perform this task). Finally, with the leaf attached the node and its internal structures populated with the node's data, we add this leaf object's pointer to the tree's leaf list for easier access and cleanup. We implemented the CBaseTree::AddLeaf method earlier to take care of adding a leaf pointer to the tree's leaf list.

At this point the leaf has been created and the node's child pointers will still be NULL, and that is how they should remain. This is an end of branch of the tree and we can simply return because our terminal node has been created. This halts the recursive process down this particular branch of the tree.

```
        // Store pointer to leaf in the node and the bounds
        pNode->Leaf = pLeaf;

        // Store the bounding box in the leaf
```

```
        pLeaf->SetBoundingBox( pNode->BoundsMin, pNode->BoundsMax );

        // Store the leaf in the leaf list
        AddLeaf( pLeaf );

        // We have reached a leaf
        return true;

    } // End if reached stop code
```

If we reach this point in the code, then it means we are not in a leaf node and as such, we need to divide the node's volume into four quadrants so that we can build appropriate polygon and detail area lists as well as new bounding volumes.

These four quadrants describe the bounding volumes of the four child nodes we will create and attach to this node.

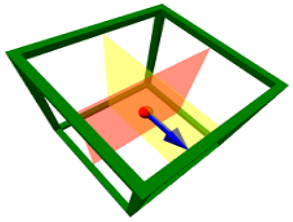


Figure 14.78

As discussed earlier in the lesson, we will determine which quadrant each polygon is in by creating two split planes that are aligned with the world X and Z axes and pass through the center point of the current node's bounding volume. The two split planes are shown in Figure 14.78. We will classify each polygon in this node's list against these two planes to determine in which quadrants they exist and to which child polygon list they should be assigned.

The next section of code generates the two split planes and stores them in the two element local Planes array. Plane[0] has its normal pointing in the direction of the Z axis (XY plane) and Plane[1] is aligned with the world YZ plane with its normal pointing down the X axis. The point on each plane which helps fully describe the plane to the D3DXPlaneFromPointNormal function is the same for each plane: the center point of the node's volume calculated by adding its maximum extents to its minimum extents and dividing the result by two. The D3DXPlaneFromPointNormal function will return a D3DXPLANE structure that describes the plane, not in point/normal format, but in normal/distance format.

```
    // Generate the two split planes
    D3DXPlaneFromPointNormal( &Planes[0], &((BoundsMin + BoundsMax) / 2.0f),
                                          &D3DXVECTOR3( 0.0f, 0.0f, 1.0f ) );

    D3DXPlaneFromPointNormal( &Planes[1], &((BoundsMin + BoundsMax) / 2.0f),
                                          &D3DXVECTOR3( 1.0f, 0.0f, 0.0f ) );
```

> **Note:** Because we have to give some descriptive names to the quadrants of our node to make things a little easier to explain, we will refer to each quadrant using a name that is relative to an imaginary individual positioned at the center of the node looking in the direction of Plane[0]'s normal. As Plane[0] faces down the positive Z axis, the quadrants behind this plane will be referred to as 'BehindLeft' and 'BehindRight'. The two quadrants in front of Plane[0] will be labelled 'InfrontLeft' and 'InfrontRight'.

If we were only supporting non-clipped trees, then our next task would be to loop through each polygon in the node's list, classify it against both planes and use the two results to determine which quadrants the polygon falls into. The polygon would then have its pointer added to the appropriate lists for those quadrants (remember that we allocated an array of four empty polygon lists on the stack which we will

fill in this function with the polygon contained in each quadrant). However, if the m_bAllowSplits member is set to true, it means the application would like this tree built so that no polygon is ever spanning node boundaries. As such, all the polygons in the node's list that are spanning any of the two node planes should be split. The original polygon that was spanning the plane should then be deleted from the list and replaced with the two new polygons that it was split into.

The next section of code is executed only in the case when a clipped tree is being built. It loops through each polygon in this node's list and classifies each polygon against the current plane being tested. If a polygon is found to be spanning the current plane we are testing, we split it into two, delete it, set its pointer entry in the list to NULL and then add the two new child polygons to the list. These child polygons will possibly be split again when the second plane is tested. We discussed code almost identical to this at the beginning of this lesson, so you should have no problem understanding how it works. Notice that the task is made very easy due the functions we have previously written (CPolygon::Split and CCollision::PolyClassifyPlane).

```
    // Split all polygons against both planes if required
    if ( m_bAllowSplits )
    {
        for ( i = 0; i < 2; ++i )
        {
            for ( PolyIterator = PolyList.begin();
                  PolyIterator != PolyList.end(); ++PolyIterator )
            {
                // Store current poly
                CurrentPoly = *PolyIterator;
                if ( !CurrentPoly ) continue;

                // Classify the poly against the first plane
                Location[0] = CCollision::PolyClassifyPlane
                              (
                                 CurrentPoly->m_pVertex,
                                 CurrentPoly->m_nVertexCount,
                                 sizeof(CVertex),
                                 (D3DXVECTOR3&)Planes[i],
                                 Planes[i].d
                              );


                if ( Location[0] == CCollision::CLASSIFY_SPANNING )
                {
                    // Split the current poly against the plane,
                    // delete it and set it to NULL in the list
                    CurrentPoly->Split( Planes[i], &FrontSplit, &BackSplit );
                    delete CurrentPoly;
                    *PolyIterator = NULL;

                    // Add these to the end of the current poly list
                    PolyList.push_back( FrontSplit );
                    PolyList.push_back( BackSplit );

                } // End if Spanning

            } // Next Polygon
```

```
        } // Next Plane

    } // End if allow splits
```

At this point, if we are building a clipped tree, any polygons in the list that were spanning the node planes will have been deleted and replaced with polygons that fit neatly into each leaf. If this is not a clipped tree, then it does not matter if a polygon is spanning a node plane as we will just assign it to all the child node lists in which it belongs.

Our next task is to loop through each polygon in the list and classify it against both of the node's split planes. We store the classification results in a local ClassifyType array called Location.

```
    // Classify the polygons and sort them into the four child lists.
    for ( PolyIterator = PolyList.begin();
          PolyIterator != PolyList.end(); ++PolyIterator )
    {
        // Store current poly
        CurrentPoly = *PolyIterator;
        if ( !CurrentPoly ) continue;

        // Classify the poly against the planes
        Location[0] = CCollision::PolyClassifyPlane( CurrentPoly->m_pVertex,
                                                     CurrentPoly->m_nVertexCount,
                                                     sizeof(CVertex),
                                                     (D3DXVECTOR3&)Planes[0],
                                                     Planes[0].d );

        Location[1] = CCollision::PolyClassifyPlane( CurrentPoly->m_pVertex,
                                                     CurrentPoly->m_nVertexCount,
                                                     sizeof(CVertex),
                                                    (D3DXVECTOR3&)Planes[1],
                                                     Planes[1].d );
```

At this point we know the current polygon's relationship to both the split planes and have the results stored, so we can figure out which quadrant it is in and which of the child polygon lists to add it to. Location[0] tells us whether it is in the back or front halfspace of the node, while Location[1] tells us in which halfspace within that halfspace (left or right) the polygon belongs.

Since the first plane split plane is aligned to the world XY plane, we know that if it is spanning the plane (only possible in the non-clipped tree case) or it is behind this plane, then it will need to have its pointer assigned to either the BehindLeft or BehindRight lists. Once we know it is behind the XY plane, we then test its classification against the YZ plane. If it is behind this plane then it must be in the BehindLeft quadrant; otherwise it must be in the BehindRight quadrant. In either case, we add it to the relevant list. ChildList[0] will contain the polygons found to be in the BehindLeft quadrant and ChildList[1] is assigned polygons that are contained in the BehindRight quadrant.

```
        // Position relative to XY plane
        if ( Location[0] == CCollision::CLASSIFY_BEHIND ||
             Location[0] == CCollision::CLASSIFY_SPANNING )
        {
```

```
            // Position relative to ZY plane
            if ( Location[1] == CCollision::CLASSIFY_BEHIND ||
                 Location[1] == CCollision::CLASSIFY_SPANNING )
                                ChildList[0].push_back( CurrentPoly );

            if ( Location[1] == CCollision::CLASSIFY_INFRONT ||
                 Location[1] == CCollision::CLASSIFY_ONPLANE ||
                 Location[1] == CCollision::CLASSIFY_SPANNING )
                                ChildList[1].push_back( CurrentPoly );

        } // End if behind or spanning
```

Notice in the above code that if a polygon is found to lie on the plane YZ plane we assign it to the node in front of the plane. An on plane polygon is essentially on the border between two nodes and we need to assign it somewhere. It does not really matter which node we choose to assign it to because if that polygon is visible, it would mean that the leaves both in front and behind that node would always be collected anyway. Likewise, if a collision query is performed and the swept sphere did intersect that polygon, it would automatically mean that it is spanning those leaves and those leaves would be collected.

If the above code was not executed it must mean the polygon is in front of the XY node plane (Plane[0]), so we must include similar code that will add the polygon to either front left or right quadrants. However, the next section of code is also executed in the spanning case as well, just as the above code was. This means that in the case of a non-clipped tree where we have a polygon spanning the XY plane, both code blocks will be executed and the polygon will be added to nodes both in front and behind this plane. This is correct, because if a polygon spans the boundaries of two nodes (which is only possible at this point in the non-clipped tree), we wish to add it to both nodes.

The next section of code handles the list assignments for polygons that are either spanning or are contained in the front halfspace of the XY plane. It is executed if the polygon is in front of the XZ plane or (as discussed above) if the polygon is on the plane. For our spatial trees, we can safely treat the on plane case as being identical to the in front case.

```
        if ( Location[0] == CCollision::CLASSIFY_INFRONT ||
              Location[1] == CCollision::CLASSIFY_ONPLANE ||
             Location[0] == CCollision::CLASSIFY_SPANNING )
        {
            // Position relative to ZY plane
            if ( Location[1] == CCollision::CLASSIFY_BEHIND ||
                 Location[1] == CCollision::CLASSIFY_SPANNING )
                                ChildList[2].push_back( CurrentPoly );

            if ( Location[1] == CCollision::CLASSIFY_INFRONT ||
                 Location[1] == CCollision::CLASSIFY_ONPLANE ||
                 Location[1] == CCollision::CLASSIFY_SPANNING )
                                 ChildList[3].push_back( CurrentPoly );

        } // End if in-front or on-plane

    } // Next Triangle
```

We can see above that ChildList[2] is the list compiled for the quadrant that is in front of the XY plane and behind the YZ plane (the back left quadrant) and ChildList[3] is for polygons that are in front of both planes (the back right quadrant).

At this point we have four polygon lists (one for each quadrant) and we now need to do the same thing with the node detail areas. Notice that the following code is identical to that we saw above with the exception that we are using our new CCollision::AABBClassifyPlane method to calculate the classification between the AABB and each plane.

```
    // Classify the areas and sort them into the child lists.
    for ( AreaIterator = AreaList.begin();
          AreaIterator != AreaList.end(); ++AreaIterator )
    {
        // Store current area
        TreeDetailArea * pDetailArea = *AreaIterator;
        if ( !pDetailArea ) continue;

        // Classify the area against the planes
        Location[0] = CCollision::AABBClassifyPlane( pDetailArea->BoundsMin,
                                                     pDetailArea->BoundsMax,
                                                     (D3DXVECTOR3&)Planes[0],
                                                     Planes[0].d );

        Location[1] = CCollision::AABBClassifyPlane( pDetailArea->BoundsMin,
                                                     pDetailArea->BoundsMax,
                                                     (D3DXVECTOR3&)Planes[1],
                                                     Planes[1].d );
```

Once again, the same logic is used to determine in which quadrant the AABB is contained and to which list it should be added.

```
        // Position relative to XY plane
        if ( Location[0] == CCollision::CLASSIFY_BEHIND ||
             Location[0] == CCollision::CLASSIFY_SPANNING )
        {
            // Position relative to ZY plane
            if ( Location[1] == CCollision::CLASSIFY_BEHIND ||
                 Location[1] == CCollision::CLASSIFY_SPANNING )
                            ChildAreaList[0].push_back( pDetailArea );

            if ( Location[1] == CCollision::CLASSIFY_INFRONT ||
                 Location[1] == CCollision::CLASSIFY_SPANNING )
                            ChildAreaList[1].push_back( pDetailArea );

        } // End if behind or spanning
        if ( Location[0] == CCollision::CLASSIFY_INFRONT ||
             Location[0] == CCollision::CLASSIFY_SPANNING )
        {
            // Position relative to ZY plane
            if ( Location[1] == CCollision::CLASSIFY_BEHIND ||
                 Location[1] == CCollision::CLASSIFY_SPANNING )
                            ChildAreaList[2].push_back( pDetailArea );
```

```
            if ( Location[1] == CCollision::CLASSIFY_INFRONT ||
                 Location[1] == CCollision::CLASSIFY_SPANNING )
                                ChildAreaList[3].push_back( pDetailArea );

        } // End if in-front or on-plane

    } // Next Detail Area
```

We are very nearly done. We have four lists of polygons that describe the polygons that fit in each quadrant of the node, and we have four lists of detail objects that also are contained (or partially contained) in each quadrant. We know at this point that we want each quadrant in this node to be represented by four new child nodes. As the BuildTree function will be called for each of these nodes and expects to be passed the node's bounding volume, we will have to create the AABB for each child node in this function.

In the next and final section of code, we set up a loop that will iterate for each child. Depending on which iteration of the loop we are on we will calculate the bounding box for that node and store it in temporary vectors NewBoundsMin and NewBoundsMax. The box computation is simple since the mid-point of the parent node will describe the location where all child volumes meet in the center of the node. We can see for example that in the first iteration of the loop we are building the volume for the BehindLeft child. Its minimum extents vector is simply the minimum extents vector of the parent node and its maximum extents vector is in the center of the parent node along the x and z axes. All quad-tree nodes inherit their height from the root node, so we can see that all nodes will always have a maximum y extents component of BoundsMax.y and will always have a minimum y extents component of BoundsMin.y. Once we have the bounding box for the current node we are processing, we allocate a new CQuadTreeNode structure and store its pointer in the parent node's child pointer list. The function then calls itself recursively passing in the new child node, the bounding box of that node that we have just calculated, and the polygon and detail area lists we compiled for its quadrant.

```
    // Build each of the children here
    for( i = 0; i < 4; ++i )
    {
        // Calculate child bounding box values
        D3DXVECTOR3 NewBoundsMin,
                    NewBoundsMax,
                    MidPoint = (BoundsMin + BoundsMax) / 2.0f;
        switch( i )
        {
            case 0: // Behind Left
                NewBoundsMin = BoundsMin;
                NewBoundsMax = D3DXVECTOR3( MidPoint.x, BoundsMax.y, MidPoint.z);
                break;

            case 1: // Behind Right
                NewBoundsMin = D3DXVECTOR3( MidPoint.x, BoundsMin.y, BoundsMin.z );
                NewBoundsMax = D3DXVECTOR3( BoundsMax.x, BoundsMax.y, MidPoint.z);
                break;

            case 2: // Infront Left
                NewBoundsMin = D3DXVECTOR3( BoundsMin.x, BoundsMin.y, MidPoint.z );
                NewBoundsMax = D3DXVECTOR3( MidPoint.x, BoundsMax.y, BoundsMax.z );
                break;
```

```
            case 3: // Infront Right
                NewBoundsMin = D3DXVECTOR3( MidPoint.x, BoundsMin.y, MidPoint.z );
                NewBoundsMax = BoundsMax;
                break;

        } // End Child Type Switch

        // Allocate child node
        pNode->Children[i] = new CQuadTreeNode;
        if ( !pNode->Children[i] ) return false;

        // Recurs into this new node
        BuildTree( pNode->Children[i],
                   ChildList[i],
                   ChildAreaList[i],
                   NewBoundsMin,
                   NewBoundsMax );

        // Clean up
        ChildList[i].clear();
        ChildAreaList[i].clear();

    } // Next Child

    // Success!
    return true;
}
```

In each iteration of the node generation loop above, the BuildTree function is passed a child node. As this function will perform all the same steps on the child node, and so on down the tree, until a leaf is located, we can imagine that once this function returned for the first child node, the entire branch of the tree that starts at that node will have been created. We then process the remaining three nodes in the loop and our job is done. Notice at the bottom of the node loop that once the BuildTree method returns for a given child, the polygon list and detail area list we compiled for it earlier in the function are no longer needed, so we release them. When this loop exits, every node and leaf under the current node being processed will have been built, so we can return. If we are in the instance of the BuildTree method that was called for the root node for example, the tree will have been completely built and program flow will pass back to the Build method which we saw previously.

You have just written a quad-tree compiler and implemented all the steps involved in the build process. The really nice thing about this is that the build functions for our oct-tree and kD-tree are almost identical. The only difference is that they split the node with a different number of planes and create a different number of child nodes. This means that we will be able to examine the build methods of the other tree with very little explanation.

Although we have covered all the code involved in the build process, there are still several query methods we must implement in the derived class to allow the application (or any user) to collect leaves based on intersections. We will examine these queries next before moving on to the other tree types.

14.17.3 The Quad-Tree Query Methods

The query functions for any spatial tree are what make the tree useful. The CollectLeavesAABB method for example is used by the application to link dynamic objects to leaves. The application can simply pass the bounding volume of the dynamic object into the CollectLeavesAABB method and have a list filled with pointers to all the leaves the volume intersected. This is obviously very handy for tasks like visibility processing since the dynamic object only has to be rendered if the IsVisible method of one of these leaves returns true.

The CollectLeavesAABB method is not just used by the application. Recall that CBaseTree demands that it is implemented in the derived class because it is called to speed up the T-junction repair process. For each polygon currently being repaired, it was used to return a list of leaves whose volumes intersect the AABB of the polygon. Only the polygons contained in these leaves would need to be tested against each other at the edge/vertex level.

For each of our tree classes we will implement two query functions which allow us to collect leaves from the tree using different primitives. The CollectLeavesAABB method sends an AABB down the tree and adds the pointer of any leaf it intersects to the passed list. The CollectLeavesRay method passes a ray down the tree and adds the pointers to any leaves the ray intersects to the passed leaf list. Fortunately, both functions are quite small due to the fact that they use the collision routines we added to CCollision earlier in this lesson. You should feel free to add query routines of your own for different primitive types (spheres would be another good choice) and extend the ISpatialTree interface as needed.

CollectLeavesAABB – CQuadTree

The CollectLeavesAABB method is a wrapper around the first call to the CollectAABBRecurse method (the recursive method that steps through the tree and adds any leaves it encounters to the passed leaf list). The CollectLeavesAABB method just starts the recursive process at the root node of the quad-tree. Here is the code.

```
bool CQuadTree::CollectLeavesAABB( LeafList & List,
                                   const D3DXVECTOR3 & Min,
                                   const D3DXVECTOR3 & Max )
{
    // Trigger Recursion
    return CollectAABBRecurse( m_pRootNode, List, Min, Max );
}
```

The method is passed three parameters. The first should be of type LeafList, which you will recall is a type definition for an STL linked list that contains ILeaf pointers. The list is passed by reference and is assumed to be empty, since it will be the job of the CollectAABBRecurse method to fill it with leaves. The second and third parameters are the extents of the AABB we would like to query the tree with (e.g., the AABB of a dynamic object).

When the CollectAABBRecurse method returns, the leaf list will contain all the pointers to leaves that were intersected by the passed AABB. Of course, the core of the query process is found inside the CollectAABBRecurse method, which we will examine next.

CollectAABBRecurse - CQuadTree

This function is passed a pointer to the node it is currently visiting, which will be a pointer to the root node the first time it is called by CollectLeavesAABB. It is also passed a leaf list which it should fill with any leaves found to be intersecting the query volume described by parameters Min and Max. The function also accepts a final boolean parameter (set to false by default) called bAutoCollect. Because this parameter is optional, and is not passed by the previous function, this will always be set to zero when it is first called for the root node. This boolean will be tracked as the tree is traversed and will be set to true once we find a node that is completely contained in the query volume.

You will recall from earlier in the lesson that we wrote a CCollision::AABBIntersectAABB method which took a boolean reference as its first parameter. The function returns true if the two boxes intersect but the boolean passed in the first parameter will also be set to true if the second box is fully contained in the first. We use this to optimize our leaf collection process because, if we find at any node in the tree that has its box fully contained inside the query volume, there is no point in performing the same AABBIntersectAABB test as we visit each of its children. Since all the children will be contained in the parent node's box, which is itself fully contained in the query volume, we know that all nodes and leaves under that node must be contained in the query volume as well. Therefore, as soon as the boolean is set to true, any further AABB tests on the children of that node will be abandoned; we will simply traverse down that branch of the tree and add leaves to the list as we find them.

The function uses the CCollision::AABBIntersectAABB method we wrote earlier in this lesson to test if the passed AABB intersects the AABB of the node currently being visited. If it does not, then this node and all of its child cannot possibly intersect the query volume, so we can return false immediately and stop traversing down this branch of the tree. Again, we only do the AABB intersection test if the bAutoCollect parameter is not set to true. If it is, then we can just proceed to collect leaves without further testing.

```
bool CQuadTree::CollectAABBRecurse( CQuadTreeNode * pNode,
                                    LeafList & List,
                                    const D3DXVECTOR3 & Min,
                                    const D3DXVECTOR3 & Max,
                                    bool  bAutoCollect /*=false*/ )
{
    bool  bResult = false;
    ULONG i;

    // Validate parameters
    if ( !pNode ) return false;

    // Does the specified box intersect this node?
    if ( !bAutoCollect && !CCollision::AABBIntersectAABB( bAutoCollect,
                                                          Min,
```

```
                                                                Max,
                                                                pNode->BoundsMin,
                                                                pNode->BoundsMax,
                                                                false,
                                                                true,
                                                                false ) ) return false;
```

Notice that we pass our boolean variable as the first parameter to the AABB intersection method so that the function can set its value to true in the event of box 2 being completely contained in box 1. As this boolean will be passed into the child node traversals this will allow us to avoid needlessly performing AABB tests at those child nodes. The extents vectors of the query volume are passed as the next two parameters followed by the extents of the node's bounding box in the second pair of parameters. Remember that the three boolean parameters at the end of this method's parameter list indicate whether we would like to ignore any axes during the test. For a quad-tree, where every node has the same height, we pass true as the bIgnoreY parameter so that the AABB intersection test is performed only on the XZ plane. It assumes that the quad-tree nodes are infinitely tall since there is no vertical partitioning in a vanilla quad-tree.

If the above collision code did not force an early return from the function, it means the query volume does intersect the node's volume. If the node's Leaf member is not set to NULL then this node is a leaf node and we add its leaf pointer to the passed list. As a leaf is a terminal node, it has no children for us to traverse and as such, our job is done and we can return.

```
    // Is there a leaf here, add it to the list
    if ( pNode->Leaf ) { List.push_back( pNode->Leaf ); return true; }
```

If the above condition was not true then it means the node intersects the query volume but the node is not a leaf node. In this case, we want to traverse into each of its four children. The function recursively calls itself for each child, making sure that it passes the original list and query volume extents down to the children. Notice in the following code how we pass the bAutoCollect boolean value into the child nodes so that if it is set to true, the AABB tests will not be performed on the child nodes (they are assumed to be inside the query volume). The remainder of the function is shown below.

```
    // Traverse down to children
    for ( i = 0; i < 4; ++i )
    {
        if ( CollectAABBRecurse( pNode->Children[i], List, Min, Max, bAutoCollect))
        bResult = true;

    } // Next Child

    // Return the 'was anything added' result.
    return bResult;
}
```

It is hard to believe that such a small function could be so powerful and useful. But thanks to spatial partitioning this is indeed the case. The function to query a ray against the tree is equally as simple, as we will see next.

CollectLeavesRay – CQuadTree

The CollectLeavesRay method can be called by the application to fill a list of all the leaves intersected by the passed ray. The method is a wrapper around the call to the CollectRayRecurse method for the root node. It is the CollectRayRecurse method which performs the traversal and collision logic.

```
bool CQuadTree::CollectLeavesRay( LeafList & List,
                                  const D3DXVECTOR3 & RayOrigin,
                                  const D3DXVECTOR3 & Velocity )
{
    return CollectRayRecurse( m_pRootNode, List, RayOrigin, Velocity );
}
```

The method is passed three parameters, the leaf list that we would like to have filled with leaves which intersect the ray, the ray origin, and the ray delta vector.

CollectRayRecurse – CQuadTree

This method is almost identical to the CollectLeavesRecurse method discussed above. It recursively calls itself visiting each node. At the current node being visited it performs a ray/box intersection test between the passed ray and the node's AABB. We use our new CCollision::RayIntersectAABB method for this and once again pass in a boolean parameter to indicate that we would like to ignore the y dimensions in the test and assume the box to be infinitely high. Although the RayIntersectAABB method returns the *t* value of intersection in its fourth parameter, we are only interested in a true or false result in this case and as such, the *t* value is ignored.

```
bool CQuadTree::CollectRayRecurse(  CQuadTreeNode * pNode,
                                    LeafList & List,
                                    const D3DXVECTOR3 & RayOrigin,
                                    const D3DXVECTOR3 & Velocity )
{
    bool  bResult = false;
    ULONG i;
    float t;

    // Validate parameters
    if ( !pNode ) return false;

    // Does the ray intersect this node?
    if ( !CCollision::RayIntersectAABB( RayOrigin, Velocity,
                                        pNode->BoundsMin,
                                        pNode->BoundsMax,
                                        t,
                                        false,
                                        true,
                                        false ) ) return false;

    // Is there a leaf here, add it to the list
```

```
    if ( pNode->Leaf ) { List.push_back( pNode->Leaf ); return true; }

    // Traverse down to children
    for ( i = 0; i < 4; ++i )
    {
        if ( CollectRayRecurse( pNode->Children[i],
                                List,
                                RayOrigin,
                                Velocity ) ) bResult = true;

    } // Next Child

    // Return the 'was anything added' result.
    return bResult;
}
```

As you can see, if the ray does not intersect the box, then it also means it cannot possibly intersect any children of the node, so we can return false are reject the rest of this branch of the tree. If the ray does intersect the box and the node is a leaf, we add its pointer to the leaf list and return. Finally, if the node is intersected by the ray but it is not a leaf node, we traverse into each of the node's children.

14.17.4 The Quad-Tree DebugDraw Routines

As discussed earlier, we have implemented some deliberately simple debug drawing routines that allow an application to request that the bounding boxes of the leaf nodes be rendered. These routines are far from optimal and are not designed to ship in commercial code. They are written help you solve any problems or choose a nice leaf size for your tree. These functions are small since we included the code that renders a box in a CBaseTree method so that all derived classes can easily implement their debug routines. We also implemented a screen tint function in CBaseTree which is called to tint the screen red when the camera enters a leaf that contains polygon or detail area data.

DebugDraw - CQuadTree

The only debug routine the application has to call is the DebugDraw routine. It should be called only after the entire scene has been rendered. It is passed the camera the application is currently using, which it passes along to the recursive function, DebugDrawRecurse. It is this function which walks the tree and renders the bounding boxes. The code to the CQuadTree::DebugDraw method is shown below.

```
void CQuadTree::DebugDraw( CCamera & Camera )
{
    // Simply start the recursive process
    DebugDrawRecurse( m_pRootNode, Camera, false );

    if ( DebugDrawRecurse( m_pRootNode, Camera, true ) )
         DrawScreenTint( 0x33FF0000 );
}
```

Notice that the DebugDrawRecurse method is called twice since we wish to render the boxes in two passes through the hierarchy. In the first pass, we want to render all the leaf nodes the camera is not currently contained in, which is why we pass false as the final parameter. These leaf nodes will be rendered blue, with Z testing enabled. This means the rendered boxes will be obscured by any geometry that is closer to the camera, just like normal scene rendering. The second time the method is called we pass true as this final parameter, which means we are only interested in rendering the leaf the camera is currently contained in. For these leaves, Z testing will be disabled so that the lines of the box overwrite anything already in the frame buffer, even if that geometry is closer to the camera. This allows us to always see all the lines of the bounding box of the leaf in which you are currently contained. If the current camera leaf has no geometry or detail areas assigned to it (it is an empty leaf) the box is rendered in green and the function return false. If it does have geometry or detail area data, the box lines are rendered in red and the function returns true. We can see in the above code that when this is the case, we also call the CBaseTree::DrawScreenTint function (covered earlier in the lesson) to tint the screen red (alpha value of 0x33).

DebugDrawRecurse - CQuadTree

This method visits each node in the tree and determines whether its box should be rendered and if so, in what color. If the boolean parameter is set to false, the function will render a leaf box when the camera position is not contained in it. If the boolean parameter is set to true, the method will traverse the tree searching only for the camera leaf node and render that box in a different manner than the other boxes. The method will also return true if the boolean parameter is set to true and the camera is contained in a leaf that has either polygon data or detail area assigned to it.

As this method is only interested in rendering leaf nodes, it first tests to see if the node it is currently visiting is a leaf node. If it is, then it uses our new CCollision::PointInAABB test to see if the camera position is contained inside the AABB of the leaf. We once again pass true for the bIgnoreY parameter in the quad-tree case so that the test is essentially performed using only the x and z components of the box and the point. We store the result of this test in the bInLeaf local variable. It will be set true only if the node this function is currently visiting is a leaf and the camera position is within its AABB.

```
bool CQuadTree::DebugDrawRecurse( CQuadTreeNode * pNode,
                                  CCamera & Camera,
                                  bool bRenderInLeaf )
{
    ULONG i;
    bool  bDrawTint = false;

    // Render the leaf
    if ( pNode->Leaf )
    {
        ILeaf * pLeaf = pNode->Leaf;
        bool bInLeaf = CCollision::PointInAABB( Camera.GetPosition(),
                                                pNode->BoundsMin,
                                                pNode->BoundsMax,
                                                false, true, false );
```

At this point, if bInLeaf is set to true and the bRenderInLeaf parameter was also set to true, it means this method is in the mode that renders only the leaf the camera is in. As bInLeaf is true, we have found that leaf so we must render it. We first fetch the polygon count and the detail area count from the leaf. If both are set to zero then it means the camera is in an empty leaf and we set its color to green. Otherwise, we set the color of the box lines to red. If the leaf is not empty, we also set the bDrawTint value to true. This is the value that will be returned from the function. It is set to false at the start of the function by default, so this function will only ever return true if the camera is in a non-empty leaf node and if the function is in the mode that renders only the camera leaf. We then call the CBaseTree::DrawBoundingBox method to render the lines of the leaf's box.

```
        if ( bRenderInLeaf && bInLeaf )
        {
            ULONG Color = 0xFFFF0000; // Red by default

            // Is there really anything in this?
            if ( pLeaf->GetPolygonCount() == 0
                 && pLeaf->GetDetailAreaCount() == 0 )
                Color = 0xFF00FF00;
            else
                bDrawTint = true;

            // Draw the bounding box
            DrawBoundingBox( pNode->BoundsMin, pNode->BoundsMax, Color, false );

        } // End if we should draw a red (inside) box here
```

Notice in the above call to DrawBoundingBox that we pass in the extents of the node's bounding box and the color we calculated based on the leaf's empty status. We also pass in false as the final parameter which instructs the method to render this box without Z testing enabled. This means all the visible lines of the box (those in the frustum) will always be rendered over the top of anything contained in the frame buffer. This will make it much easier to see the extents of the box the camera is currently in without worrying about the lines of the box being obscured by nearby geometry.

We have handled the case for when the function is in 'draw camera leaf only' mode. The next section of code is executed for the 'draw everything except the camera leaf' mode and the current node is not a node the camera is contained in. It uses the CBaseTree::DrawBoundingBox method to draw a blue box, only this time we pass true as the final parameter so that the depth buffer is enabled and the box is rendered in the normal way.

```
        // Draw blue box?
        if ( !bRenderInLeaf && !bInLeaf )
        {
            DrawBoundingBox( pNode->BoundsMin, pNode->BoundsMax, 0xFF0000FF,true );

        } // End if we should draw a blue box (outside) here
```

Finally, as detail areas are not something we can usually see, we will also render a box around any detail areas that might be contained in the current leaf node. As you can see, we simply set up a loop to extract each detail area assigned to the current leaf and render a blue box using the detail area's bounding box. For detail areas, we render with depth testing disabled so that they are not obscured by nearby geometry.

```
        for ( i = 0; i < pLeaf->GetDetailAreaCount(); ++i )
        {
            TreeDetailArea * pDetailArea = pLeaf->GetDetailArea( i );
            if ( !pDetailArea ) continue;

            // Now let's render the detail area
            DrawBoundingBox( pDetailArea->BoundsMin,
                             pDetailArea->BoundsMax,
                             0xFF00FF00,
                             false );

        } // Next Detail Area

    } // End if not a leaf
```

We have now seen all the code that is executed if the current node is a leaf node. If it is not, then this is a non-terminal node that will have four children, so we better visit those too.

```
    // Step down to children.
    for ( i = 0; i < 4; ++i )
    {
        if ( pNode->Children[i] )
        {
            if ( DebugDrawRecurse( pNode->Children[i], Camera, bRenderInLeaf ) )
                 bDrawTint = true;

        } // End if child exists

    } // Next child

    // Return whether we should draw the tint
    return bDrawTint;

}
```

If this function returns true, it means the camera is in a leaf that contains geometry or detail areas and the parent function (DebugDraw) tints the screen slightly red when this is the case.

14.17.5 Quad-Tree Conclusion

We have now covered all the code to build and query the quad-tree for collision and spatial queries. In the next lesson we will add code to CBaseTree (and one or two small functions to the derived classes) that will exist to implement an efficient hardware rendering system. The focus of this lesson however has been on using the tree for spatial queries and we have now successfully created a quad-tree. We will now move on to discuss the implementation of the other tree types, which should be quite easy now that we have covered the quad-tree code and the concepts are all very similar.

14.18 The Y-Variant Quad-Tree

An alternative approach to working with quad-trees is to factor in the Y component at each node such that the bounding box has a better fit around the geometry set (i.e., it is not assumed to be infinitely tall). Figure 14.79 reminds us what a Y-variant quad-tree node might look like with each of its children not filling the entire volume of the parent node.

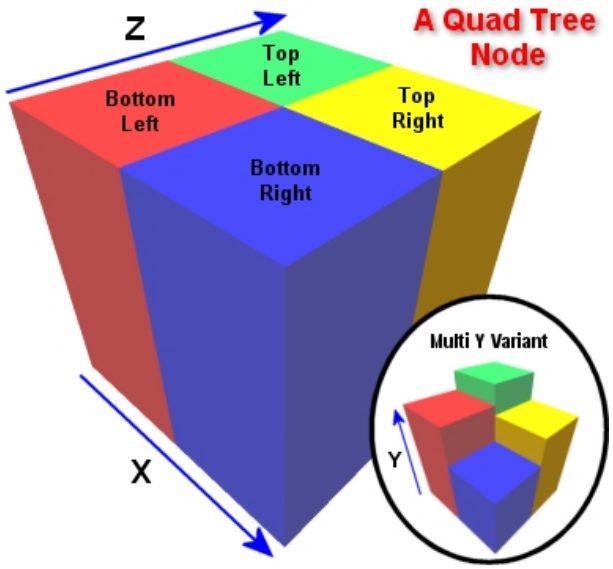


Figure 14.79

Because the bounding boxes are now built to the fit the size of the geometry contained at each node along the vertical extent, we get nodes with smaller boxes that provide more accurate querying versus the vanilla quad-tree case.

We can imagine a situation for example where a spacecraft might be hovering in the skies above a cityscape. In the vanilla quad-tree case, the leaf in which the cityscape is contained would be infinitely tall (or as tall as the tallest thing in the scene) and as such, the spacecraft would be considered to be in the same leaf node as the cityscape even though it was far above it. This would lead to the cityscape being rendered even if the plane was so far above it that that the geometry of the cityscape was well below the frustum of the camera mounted on that craft. In the Y-variant quad-tree case, the leaf in which the cityscape is contained would only be as tall as the tallest building in that city, which would mean the spacecraft would be outside that leaf and it would not be rendered. Of course, the savings are a lot less obvious when we are compiling indoor scenes.

While an oct-tree is one obvious way to solve this problem, we can make a modification to the quad-tree implementation and derive a new class called CQuadTreeYV that will be built slightly differently. At each node, the bounding box's Y extents are not inherited from the root node, but are calculated based on the polygon data in that node. This will allow us to more effectively frustum cull leaves that might not be culled in the vanilla quad-tree implementation.

Although it might seem that the Y-variant quad-tree would also introduce similar savings in collision queries (which is technically true), we will not be able to do this with our implementation. You will recall that the CollectLeavesAABB used by our collision system is implemented in the vanilla quad-tree case to assume infinitely tall leaves by way of ignoring the Y component of the node's box during AABB/AABB intersection tests (the CollectLeavesRay method does the same). Although we could easily re-implement these in our Y-variant version of the quad-tree, this design does not cooperate well with our dynamic object system.

Looking at Figure 14.79 we can see that at any given node, if the child volumes do not totally consume the parent volume, we have a situation where locations within the quad-tree do not fall within any leaf

node. In the circular inset we can see some children have a very small Y extent while the Y extent of the parent node would be equal to the green child. What happens if we feed an AABB into the tree which happens to fall in the space just above the blue child node? It would clearly be contained inside the parent node's volume, but it would not fall into any of its children. Thus, it would essentially be in no-man's land and the function would simply return and no leaf would be added to the leaf collection list.

Now, if this was a collision query being performed, we would really benefit from the Y-variant tree's smaller boxes. If we passed the swept sphere's AABB down the tree and it ended up in an area that is not bounded by a leaf, the returned list would be empty and we would know that the swept sphere does not intersect any geometry. However, the same CollectLeavesAABB method is also used by our application to determine which leaves a dynamic object is in. The object is only rendered if at least one of its leaves is visible. However, in the Y-variant case we might find that when we feed in the dynamic object's bounding volume, it exists in no leaves. What do we do?

We would have no choice in such situations other than to always render those dynamic objects. That is, if a dynamic object is not currently in a leaf, we do not know whether or not it is visible, so we should render it to be safe. We could end up rendering many objects which are nowhere near the camera and not even close to being visible, and this would certain hinder performance in many cases. Of course, there are ways around such problems; we could store dynamic object pointers at nodes as well as leaves and when traversing the tree, render any objects that are contained at that node before moving on. However, that really does not fit our design very well, where we make a clear distinction between nodes and leaves. So we have made the design decision to ignore the y extents of a node's box during the leaf collection functions. This means that, just as in the case of the vanilla quad-tree, during the leaf collection process the node volumes are assumed to have infinite Y extents. The savings we get from implementing the YV quad-tree will be purely during the frustum culling pass of the tree for static objects.

Note that this tree will behave almost exactly like the vanilla quad-tree -- even the CollectLeavesAABB method will be the same since we are choosing to ignore the Y components. In fact, the only difference (frustum culling aside, as this will be discussed in the next lesson) between the quad-tree and the YV quad-tree will be a small section of code in the recursive BuildTree function that fits the bounding box of the node to the polygon set along the Y extents, instead of inheriting it from the root node's volume. As the BuildTree method of CQuadTree is virtual, we can simply derive CQuadTreeYV from CQuadTree and just override it.

Below we show the class declaration for CQuadTreeYV which is contained in CQuadTreeYV.h.

```
class CQuadTreeYV : public CQuadTree
{
public:

    // Constructors & Destructors for This Class.
    CQuadTreeYV( LPDIRECT3DDEVICE9 pDevice,
                 bool bHardwareTnL,
                 float fMinLeafSize = 300.0f,
                 ULONG nMinPolyCount = 600,
                 ULONG nMinAreaCount = 0 ) : CQuadTree( pDevice,
                                                        bHardwareTnL,
                                                        fMinLeafSize,
```

```
                                                        nMinPolyCount,
                                                        nMinAreaCount ) {};

protected:
    // Protected virtual Functions for This Class

    virtual bool BuildTree ( CQuadTreeNode * pNode,
                             PolygonList PolyList,
                             DetailAreaList AreaList,
                             const D3DXVECTOR3 & BoundsMin,
                             const D3DXVECTOR3 & BoundsMax );
};
```

As you can see, with the exception of the constructor which is empty and simply passes the data on to the base class (CQuadTree), we have to implement just one function. The BuildTree method is a method we are now very familiar with. We will override it in this class due to the fact that we need to calculate the bounding box of each node a little differently. We will now cover the BuildTree method and highlight the small differences in the YV version.

BuildTree – CQuadTreeYV

Although we will see the complete function code listing below, most of it is unchanged from the CQuadTree::BuildTree method. Recall that this is the method that is called from the Build method and passed the root node and its polygon and detail area data along with the bounding box of the root node. This function will then recursively build the entire tree.

Because this code has already been thoroughly discussed, we will move quickly and only stop to examine the changes from the function that it overrides.

```
bool CQuadTreeYV::BuildTree( CQuadTreeNode * pNode,
                             PolygonList PolyList,
                             DetailAreaList AreaList,
                             const D3DXVECTOR3 & BoundsMin,
                             const D3DXVECTOR3 & BoundsMax )
{
    D3DXVECTOR3                 Normal;
    PolygonList::iterator       PolyIterator;
    DetailAreaList::iterator    AreaIterator;
    CPolygon                  * CurrentPoly, * FrontSplit, * BackSplit;
    PolygonList                 ChildList[4];
    DetailAreaList              ChildAreaList[4];
    CCollision::CLASSIFYTYPE    Location[2];
    D3DXPLANE                   Planes[2];
    unsigned long               i;
    bool                        bStopCode;

    // Store the bounding box properties in the node
    pNode->BoundsMin = BoundsMin;
    pNode->BoundsMax = BoundsMax;
```

```
    // Calculate 'Stop' code
    D3DXVECTOR3 vecLeafSize = D3DXVECTOR3( BoundsMax.x - BoundsMin.x,
                                           0,
                                           BoundsMax.z - BoundsMin.z );

    bStopCode  = (AreaList.size() == 0 && PolyList.size() == 0) ||
                 (AreaList.size() <= m_nMinAreaCount &&
                  PolyList.size() <= m_nMinPolyCount) ||
                  D3DXVec3Length( &vecLeafSize ) <= m_fMinLeafSize;
```

The first section code stored the passed bounding box in the passed node and calculated the 2D size of the node (vecLeafSize). Although this is a YV quad-tree we still do not want the y extents of the node to play any part in whether or not we subdivide. Unlike the oct-tree where we intend to subdivide space vertically, the YV quad-tree does not have leaves stacked on top of each other. So a leaf must still contain the entire vertical range of geometry that exists in the XZ space. The only difference is that when we do create a node/leaf bounding box, we will calculate the extents of the actual geometry that made it into that node. After we have calculated the leaf size, we compute the bStopCode boolean. This code is all unchanged.

As before, if the stop code is set to true then we have found a node that should become a leaf so we allocate a new CBaseLeaf object. We then loop through each polygon in the passed list (the polygons that made it into this node) and add their pointers to the leaf. Remember that if this is a clipped tree, we also add the polygons to the tree's polygon list.

```
    // If we reached our stop limit, build a leaf here
    if ( bStopCode )
    {
        // Build a leaf
        CBaseLeaf * pLeaf = new CBaseLeaf( this );
        if ( !pLeaf ) return false;

        // Store the polygons
        for ( PolyIterator = PolyList.begin();
              PolyIterator != PolyList.end();
              ++PolyIterator )
        {
            // Retrieve poly
            CurrentPoly = *PolyIterator;
            if ( !CurrentPoly ) continue;

            // Add to the polygon list ONLY if splitting was allowed
            if ( m_bAllowSplits ) AddPolygon( CurrentPoly );

            // Also add a reference to the leaf's list
            pLeaf->AddPolygon( CurrentPoly );

        } // Next Polygon
```

Of course, we do the same for any detail area that made it into this leaf.

```
        // Store the area lists
        for ( AreaIterator = AreaList.begin();
```

```
              AreaIterator != AreaList.end(); ++AreaIterator )
        {
            // Retrieve detail area item
            TreeDetailArea * pDetailArea = *AreaIterator;
            if ( !pDetailArea ) continue;

            // Add a reference to the leaf's list
            pLeaf->AddDetailArea( pDetailArea );

        } // Next Polygon
```

We then assign the node's leaf pointer to point at the newly populated CBaseLeaf and set the leaf object's bounding box to that of the node it represents. We then add the leaf pointer to the tree's leaf array and return. That takes care of the leaf case.

```
        // Store pointer to leaf in the node and the bounds
        pNode->Leaf = pLeaf;

        // Store the bounding box in the leaf
        pLeaf->SetBoundingBox( pNode->BoundsMin, pNode->BoundsMax );

        // Store the leaf in the leaf list
        AddLeaf( pLeaf );

        // We have reached a leaf
        return true;

    } // End if reached stop code
```

The next piece of code is new for the Y-variant quad-tree. Remember that we only reach this part of the function if the current node is not a leaf. Although we already know the X and Z extents of the node's bounding box because they were passed into the function, the y extents will currently represent the y extents of the parent node's volume. We want to set the height of this node's volume so that it matches the y range of vertices it stores. Therefore, we first set the y extents of the node's AABB to impossibly small (inside out) values. We then loop through every polygon in the passed list (the polygons contained in this node) and test the y extents of each of its vertices against the y extents of the node's box. Whenever we find a vertex that has a y component that is outside the box, we grow the box extents to contain it. At the end of the following section of code, the current node's bounding box will be an exact fit (vertically) around the polygon data passed into this node.

```
    // Update the Y coordinate of the bounding box for the 'Y Variant'
    // quad-tree information

    pNode->BoundsMin.y = FLT_MAX; pNode->BoundsMax.y = -FLT_MAX;

    for ( PolyIterator = PolyList.begin();
          PolyIterator != PolyList.end(); ++PolyIterator )
    {
        // Store current poly
        CurrentPoly = *PolyIterator;
        if ( !CurrentPoly ) continue;
```

```
        // Loop through each vertex
        for ( i = 0; i < CurrentPoly->m_nVertexCount; ++i )
        {
            CVertex * pVertex = &CurrentPoly->m_pVertex[i];
            if ( pVertex->y < pNode->BoundsMin.y ) pNode->BoundsMin.y = pVertex->y;
            if ( pVertex->y > pNode->BoundsMax.y ) pNode->BoundsMax.y = pVertex->y;

        } // Next Vertex

    } // Next Polygon
```

Of course, we must make sure that the box is also large enough to contain any detail areas that made it into this node. In the next section of code we loop through each detail area in this node and grow the y extents of the node's AABB if any detail area is found to not be fully contained.

```
    for ( AreaIterator = AreaList.begin();
          AreaIterator != AreaList.end(); ++AreaIterator )
    {
        // Store current detail area
        TreeDetailArea * pDetailArea = *AreaIterator;

        if ( pDetailArea->BoundsMin.x < pNode->BoundsMin.x )
             pNode->BoundsMin.x = pDetailArea->BoundsMin.x;

        if ( pDetailArea->BoundsMin.y < pNode->BoundsMin.y )
             pNode->BoundsMin.y = pDetailArea->BoundsMin.y;

        if ( pDetailArea->BoundsMin.z < pNode->BoundsMin.z )
             pNode->BoundsMin.z = pDetailArea->BoundsMin.z;

        if ( pDetailArea->BoundsMax.x > pNode->BoundsMax.x )
             pNode->BoundsMax.x = pDetailArea->BoundsMax.x;

        if ( pDetailArea->BoundsMax.y > pNode->BoundsMax.y )
             pNode->BoundsMax.y = pDetailArea->BoundsMax.y;

        if ( pDetailArea->BoundsMax.z > pNode->BoundsMax.z )
             pNode->BoundsMax.z = pDetailArea->BoundsMax.z;

    } // Next Detail Area
```

With the node's bounding volume now a tight fit around its data, we create the two split planes as before.

```
    // Generate the two split planes
    D3DXPlaneFromPointNormal( &Planes[0],
                              &((pNode->BoundsMin + pNode->BoundsMax) / 2.0f),
                              &D3DXVECTOR3( 0.0f, 0.0f, 1.0f ) );

    D3DXPlaneFromPointNormal( &Planes[1],
                              &((pNode->BoundsMin + pNode->BoundsMax) / 2.0f),
                              &D3DXVECTOR3( 1.0f, 0.0f, 0.0f ) );
```

If we are building a clipped tree then we will first need to split all the polygons in this node's list to the planes so that we end up with a list of polygons that will each fit into exactly one child.

```
    // Split all polygons against both planes if required
    if ( m_bAllowSplits )
    {
        for ( i = 0; i < 2; ++i )
        {
            for ( PolyIterator = PolyList.begin();
                  PolyIterator != PolyList.end(); ++PolyIterator )
            {
                // Store current poly
                CurrentPoly = *PolyIterator;
                if ( !CurrentPoly ) continue;

        // Classify the poly against the first plane
        Location[0] = CCollision::PolyClassifyPlane( CurrentPoly->m_pVertex,
                                                     CurrentPoly->m_nVertexCount,
                                                     sizeof(CVertex),
                                                     (D3DXVECTOR3&)Planes[i],
                                                     Planes[i].d );

                if ( Location[0] == CCollision::CLASSIFY_SPANNING )
                {
                    // Split the current poly against the plane
                    CurrentPoly->Split( Planes[i], &FrontSplit, &BackSplit );
                    delete CurrentPoly;
                    *PolyIterator = NULL;

                    // Add these to the end of the current poly list
                    PolyList.push_back( FrontSplit );
                    PolyList.push_back( BackSplit );

                } // End if Spanning

            } // Next Polygon

        } // Next Plane

    } // End if allow splits
```

As before, the next section of code adds each polygon to the appropriate child list. It is unchanged from the normal quad-tree case.

```
    // Classify the polygons and sort them into the four child lists.
    for ( PolyIterator = PolyList.begin();
          PolyIterator != PolyList.end(); ++PolyIterator )
    {
        // Store current poly
        CurrentPoly = *PolyIterator;
        if ( !CurrentPoly ) continue;

        // Classify the poly against the planes
        Location[0] = CCollision::PolyClassifyPlane( CurrentPoly->m_pVertex,
                                                     CurrentPoly->m_nVertexCount,
```

```
                                                    sizeof(CVertex),
                                                    (D3DXVECTOR3&)Planes[0],
                                                    Planes[0].d );

        Location[1] = CCollision::PolyClassifyPlane( CurrentPoly->m_pVertex,
                                                    CurrentPoly->m_nVertexCount,
                                                    sizeof(CVertex),
                                                    (D3DXVECTOR3&)Planes[1],
                                                    Planes[1].d );

        // Position relative to XY plane
        if ( Location[0] == CCollision::CLASSIFY_BEHIND ||
             Location[0] == CCollision::CLASSIFY_SPANNING )
        {
            // Position relative to ZY plane
            if ( Location[1] == CCollision::CLASSIFY_BEHIND ||
                 Location[1] == CCollision::CLASSIFY_SPANNING )
                            ChildList[0].push_back( CurrentPoly );

            if ( Location[1] == CCollision::CLASSIFY_INFRONT ||
                 Location[1] == CCollision::CLASSIFY_ONPLANE ||
                 Location[1] == CCollision::CLASSIFY_SPANNING )
                             ChildList[1].push_back( CurrentPoly );

        } // End if behind or spanning

        if ( Location[0] == CCollision::CLASSIFY_INFRONT ||
             Location[0] == CCollision::CLASSIFY_ONPLANE ||
             Location[0] == CCollision::CLASSIFY_SPANNING )
        {
            // Position relative to ZY plane
            if ( Location[1] == CCollision::CLASSIFY_BEHIND ||
                 Location[1] == CCollision::CLASSIFY_SPANNING )
                            ChildList[2].push_back( CurrentPoly );

            if ( Location[1] == CCollision::CLASSIFY_INFRONT ||
                 Location[1] == CCollision::CLASSIFY_ONPLANE ||
                 Location[1] == CCollision::CLASSIFY_SPANNING )
                            ChildList[3].push_back( CurrentPoly );

        } // End if in-front or on-plane

    } // Next Triangle
```

At this point, the polygon data at this node has been sorted into four lists (one per quadrant). We now have to sort the detail areas into four lists also so that we know which children they should be assigned to. This code is also unchanged from the normal quad-tree code.

```
    // Classify the areas and sort them into the child lists.
    for ( AreaIterator = AreaList.begin();
          AreaIterator != AreaList.end(); ++AreaIterator )
    {
        // Store current area
        TreeDetailArea * pDetailArea = *AreaIterator;
        if ( !pDetailArea ) continue;
```

```
        // Classify the poly against the planes
        Location[0] = CCollision::AABBClassifyPlane( pDetailArea->BoundsMin,
                                                     pDetailArea->BoundsMax,
                                                     (D3DXVECTOR3&)Planes[0],
                                                     Planes[0].d );

        Location[1] = CCollision::AABBClassifyPlane( pDetailArea->BoundsMin,
                                                     pDetailArea->BoundsMax,
                                                     (D3DXVECTOR3&)Planes[1],
                                                     Planes[1].d );

        // Position relative to XY plane
        if ( Location[0] == CCollision::CLASSIFY_BEHIND ||
             Location[0] == CCollision::CLASSIFY_SPANNING )
        {
            // Position relative to ZY plane
            if ( Location[1] == CCollision::CLASSIFY_BEHIND ||
                 Location[1] == CCollision::CLASSIFY_SPANNING )
                                ChildAreaList[0].push_back( pDetailArea );

            if ( Location[1] == CCollision::CLASSIFY_INFRONT ||
                 Location[1] == CCollision::CLASSIFY_SPANNING )
                                ChildAreaList[1].push_back( pDetailArea );

        } // End if behind or spanning

        if ( Location[0] == CCollision::CLASSIFY_INFRONT ||
             Location[0] == CCollision::CLASSIFY_SPANNING )
        {
            // Position relative to ZY plane
            if ( Location[1] == CCollision::CLASSIFY_BEHIND ||
                 Location[1] == CCollision::CLASSIFY_SPANNING )
                                ChildAreaList[2].push_back( pDetailArea );

            if ( Location[1] == CCollision::CLASSIFY_INFRONT ||
                 Location[1] == CCollision::CLASSIFY_SPANNING )
                                ChildAreaList[3].push_back( pDetailArea );

        } // End if in-front or on-plane

    } // Next Detail Area
```

We now have four lists of polygons and four lists of detail objects, so we create the four child nodes to which they belong and calculate the bounding box of each child. As before, we set up a loop where a single child node is created with each iteration. In each of the four iterations, we allocate a new quad-tree node and store its pointer in the parent node's child list and then calculate the bounding volume of that node by assigning the associated quadrant of the parent node's bounding volume. We then step into the child node by issuing the recursive call.

```
    // Build each of the children here
    for( i = 0; i < 4; ++i )
    {
        // Calculate child bounding box values
        D3DXVECTOR3 NewBoundsMin, NewBoundsMax,
```

```
                MidPoint = (pNode->BoundsMin + pNode->BoundsMax) / 2.0f;

        switch( i )
        {
        case 0: // Behind Left
            NewBoundsMin = pNode->BoundsMin;
            NewBoundsMax = D3DXVECTOR3( MidPoint.x,
                                        pNode->BoundsMax.y,
                                        MidPoint.z);
            break;

        case 1: // Behind Right
            NewBoundsMin = D3DXVECTOR3( MidPoint.x,
                                        pNode->BoundsMin.y,
                                        pNode->BoundsMin.z );

            NewBoundsMax = D3DXVECTOR3( pNode->BoundsMax.x,
                                        pNode->BoundsMax.y,
                                        MidPoint.z);
            break;


        case 2: // InFront Left
            NewBoundsMin = D3DXVECTOR3( pNode->BoundsMin.x,
                                        pNode->BoundsMin.y,
                                        MidPoint.z );

            NewBoundsMax = D3DXVECTOR3( MidPoint.x,
                                        pNode->BoundsMax.y,
                                        pNode->BoundsMax.z );
            break;

        case 3: // InFront Right
            NewBoundsMin = D3DXVECTOR3( MidPoint.x,
                                        pNode->BoundsMin.y,
                                        MidPoint.z );

            NewBoundsMax = pNode->BoundsMax;
            break;

        } // End Child Type Switch


        // Allocate child node
        pNode->Children[i] = new CQuadTreeNode;
        if ( !pNode->Children[i] ) return false;

        // Recurse into this new node
        BuildTree( pNode->Children[i],
                   ChildList[i],
                   ChildAreaList[i],
                   NewBoundsMin,
                   NewBoundsMax );

        // Clean up
        ChildList[i].clear();
        ChildAreaList[i].clear();
```

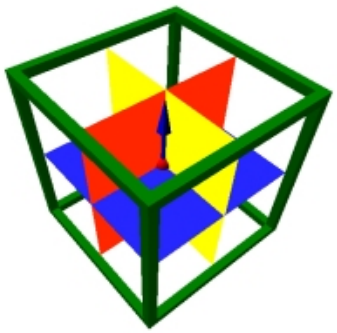
```
    } // Next Child

    // Success!
    return true;
}
```

Note how quickly we were able to create new tree behavior. In the case of the Y-variant quad-tree, it meant overriding a single method. That is all the code we need to cover for the YV Quad-tree.

14.19 Oct-Tree Implementation

The source files COctTree.cpp and COctTree.h contain the code for our oct-tree implementation. This tree is derived from CBaseTree and as such, we must implement the same methods from ISpatialTree that are not implemented in CBaseTree. Just as with the quad-tree, we will only need to implement the Build, CollectLeavesAABB, CollectLeavesRay and a few others in order to create our oct-tree because CBaseTree provides all the housekeeping tasks.

The oct-tree implementation will be so similar to that of the quad-tree that examining its code will be trivial for the most part. The only real difference in each version of the method implemented for the oct-tree is that each node has eight children to visit/create instead of four. In the Build function we will now have three planes to clip and classify against instead of two. These three planes will divide the node's volume in octants (instead of quadrants) as shown in Figure 14.80.



Octree Node Partitioning

Figure 14.80

Although CBaseTree provides the leaf structure that all our trees will use, the node structure is dependant on the tree being built. For example, in the quad-tree case we implemented the CQuadTreeNode object to represent a single node in the tree. This node has four child pointers as we would expect. The Y-variant quad-tree also shared this same node structure. The oct-tree will need its own node structure that is capable or storing pointers to eight child nodes instead of four. That is really the only difference between the node objects used by all of our tree types; the varying number of children.

14.19.1 COctTreeNode – The Source Code

Each node in our oct-tree will be represented by an object of type COctTreeNode which is declared and implemented in COctTree.h and COctTree.cpp. It contains the same SetVisible member function as its quad-tree counterpart and also stores the same members. It stores a pointer to a CBaseLeaf object that will be used only by terminal nodes and contains two vectors describing the extents of the node's bounding volume. It also contains the same LastFrustumPlane member which we will discuss in the following lesson as it is used to speed up visibility processing. Finally, you will notice that the only difference (except the name) between this object and the CQuadTreeNode object is that each node stores an array of 8 COctTreeNode pointers instead of an array of 4 CQuadTreeNode pointers.

```
class COctTreeNode
{
public:

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


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


    // Public Variables for This Class
    COctTreeNode  * Children[8];        // The eight 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;   // 'last plane' index.
};
```

SetVisible – COctTreeNode

Each of our node types supports the SetVisible method which is used by the tree to set the visibility status of all leaves below the node for which it is called. It works in exactly the same way as its quad-tree node counterpart. How this function is used will become clearer in the following lesson when we implement the rendering and visibility code for our trees.

The method is passed a boolean parameter indicating whether the node is considered visible or not. If the node is a leaf then the attached leaf object has its visibility status set to true. If this is not a leaf we simply traverse into the eight children performing the same test. As a node has no member to store its visibility status, what we are actually doing here is just traversing down to find any leaf nodes and setting their status to the parameter passed to the top level instance of the function.

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

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

    // Recurse down if applicable
    for ( i = 0; i < 8; ++i )
    {
        if ( Children[i] ) Children[i]->SetVisible(bVisible);

    } // Next Child
}
```

Notice that the only difference between this code and the quad-tree node's method is that we now have a loop that visits eight children instead of four.

14.19.2 COctTree – The Source Code

The COctTree class is derived from CBaseTree and therefore must implement oct-tree versions of the methods we implemented in CQuadTree. The class declaration is contained in COctTree.h and is shown below. Notice that it implements the exact same set of methods we had to implement for our quad-tree class.

```
class COctTree : public CBaseTree
{
public:

       // Constructors & Destructors for This Class.
       virtual ~COctTree( );
       COctTree( LPDIRECT3DDEVICE9 pDevice,
                 bool bHardwareTnL,
                 float fMinLeafSize = 300.0f,
                 ULONG nMinPolyCount = 600,
                 ULONG nMinAreaCount = 0 );


    // Public Virtual Functions for This Class (from base).
    virtual bool Build                  ( bool bAllowSplits = true );
    virtual void ProcessVisibility      ( CCamera & Camera );
    virtual bool CollectLeavesAABB      ( LeafList & List,
                                          const D3DXVECTOR3 & Min,
                                          const D3DXVECTOR3 & Max );
    virtual bool CollectLeavesRay       ( LeafList & List,
                                          const D3DXVECTOR3 & RayOrigin,
                                          const D3DXVECTOR3 & Velocity );

    virtual void DebugDraw              ( CCamera & Camera );
    virtual bool GetSceneBounds         ( D3DXVECTOR3 & Min, D3DXVECTOR3 & Max );
```

```
protected:

    // Protected virtual Functions for This Class
    virtual bool BuildTree              ( COctTreeNode * pNode,
                                          PolygonList PolyList,
                                          DetailAreaList AreaList,
                                          const D3DXVECTOR3 & BoundsMin,
                                          const D3DXVECTOR3 & BoundsMax );

    // Protected Functions for This Class
    void         UpdateTreeVisibility   ( COctTreeNode * pNode,
                                          CCamera & Camera,
                                          UCHAR FrustumBits = 0x0 );

    bool         DebugDrawRecurse       ( COctTreeNode * pNode,
                                          CCamera & Camera, bool bRenderInLeaf );

    bool         CollectAABBRecurse     ( COctTreeNode * pNode,
                                          LeafList & List,
                                          const D3DXVECTOR3 & Min,
                                          const D3DXVECTOR3 & Max,
                                          bool bAutoCollect = false );

    bool         CollectRayRecurse      ( COctTreeNode * pNode,
                                          LeafList & List,
                                          const D3DXVECTOR3 & RayOrigin,
                                          const D3DXVECTOR3 & Velocity );

    // Protected Variables for This Class
    COctTreeNode * m_pRootNode;         // The root node of the tree
    bool           m_bAllowSplits;      // Is splitting allowed?
    float          m_fMinLeafSize;      // Min leaf size stop code
    ULONG          m_nMinPolyCount;     // Min polygon count stop code
    ULONG          m_nMinAreaCount;     // Min detail area count stop code

};
```

The member variables are exactly the same in both name and purpose to those we discussed when implementing the quad-tree. The only slight difference in member variables is that the root node pointer stored by the tree is now of type COctTreeNode instead of CQuadTreeNode. The methods are also exactly the same as their quad-tree counterparts with the exception that the recursive methods that accept node parameters now accept nodes of type COctTreeNode.

Let us now cover the methods of COctTree and discuss how they differ from their quad-tree counterparts.

Build - COctTree

As we know, the Build method is the method called by the application to build the tree. It is called after all geometry and detail areas have been registered. It is not the recursive function that builds the tree but

is instead a convenient wrapper around the top level call to the recursive BuildTree method. Because the code is almost unchanged from its quad-tree counter part we will progress through it quickly.

The first bit of the function initializes the vectors that will be used to record the size of the root node's bounding box to impossibly small values before allocating a new COctTreeNode object that will become the root node of the tree. We also store the value of the boolean parameter in the m_bAllowSplits member variable so that we know during the recursive build process whether we should be clipping the polygon data at each node.

```
bool COctTree::Build( bool bAllowSplits /* = false */ )
{
    PolygonList::iterator       PolyIterator = m_Polygons.begin();
    DetailAreaList::iterator    AreaIterator = m_DetailAreas.begin();
    PolygonList                 PolyList;
    DetailAreaList              AreaList;
    unsigned long               i;

    // Reset our tree info values.
    D3DXVECTOR3 vecBoundsMin( FLT_MAX, FLT_MAX, FLT_MAX );
    D3DXVECTOR3 vecBoundsMax( -FLT_MAX, -FLT_MAX, -FLT_MAX );

    // Allocate a new root node
    m_pRootNode = new COctTreeNode;
    if ( !m_pRootNode ) return false;

    // Store the allow splits value for later retrieval.
    m_bAllowSplits = bAllowSplits;
```

Now we need to calculate the size of the root nodes bounding volume by looping through every polygon in the tree's polygon list and testing each of its vertices against the current box extents. If we find any vertex is outside these extents, we grow the box. At the end of the loop we will have a bounding box that contains all the vertices of all the polygons.

```
    // Loop through all of the initial polygons
    for ( ; PolyIterator != m_Polygons.end(); ++PolyIterator )
    {
        // Retrieve the polygon
        CPolygon * pPoly = *PolyIterator;
        if ( !pPoly ) continue;

        // Calculate total scene bounding box.
        for ( i = 0; i < pPoly->m_nVertexCount; ++i )
        {
            // Store info
            CVertex * pVertex = &pPoly->m_pVertex[i];
            if ( pVertex->x < vecBoundsMin.x ) vecBoundsMin.x = pVertex->x;
            if ( pVertex->y < vecBoundsMin.y ) vecBoundsMin.y = pVertex->y;
            if ( pVertex->z < vecBoundsMin.z ) vecBoundsMin.z = pVertex->z;
            if ( pVertex->x > vecBoundsMax.x ) vecBoundsMax.x = pVertex->x;
            if ( pVertex->y > vecBoundsMax.y ) vecBoundsMax.y = pVertex->y;
            if ( pVertex->z > vecBoundsMax.z ) vecBoundsMax.z = pVertex->z;

        } // Next Vertex
```

```
        // Store this polygon in the top polygon list
        PolyList.push_back( pPoly );

    } // Next Polygon

    // Clear the initial polygon list if we are going to split the polygons
    // as this will eventually become storage for whatever gets built
    if ( bAllowSplits ) m_Polygons.clear();
```

Just as before, at the bottom of the outer loop, as we test each polygon, we add it to the temporary PolyList. It is this list that will be passed into the recursive process. As we are currently at the root node, PolyList will contain a complete copy of all the pointers that were registered with the tree and stored in the m_Polygons list. Outside the loop you can see that just as before, if we are going to build a clipped tree, we clear the tree's polygon list since we will need to rebuild it from scratch with the clipped polygon data that makes it into each leaf node.

Our root node's bounding box is currently large enough to contain all the polygon data but we need to make sure it is large enough to contain any detail areas that may have been registered with the tree as well. In the next section of code we loop through each detail area and extend the bounding box extents if any detail area is found not to be fully contained inside it. Notice also that as we process each detail area, we also add its pointer to the temporary detail area list. This is the detail area list we will send into the recursive build process.

```
    // Loop through all of the detail areas
    for ( ; AreaIterator != m_DetailAreas.end(); ++AreaIterator )
    {
        // Retrieve the detail area
        TreeDetailArea * pDetailArea = *AreaIterator;
        if ( !pDetailArea ) continue;

        // Calculate total scene bounding box.
        D3DXVECTOR3 & Min = pDetailArea->BoundsMin;
        D3DXVECTOR3 & Max = pDetailArea->BoundsMax;
        if ( Min.x < vecBoundsMin.x ) vecBoundsMin.x = Min.x;
        if ( Min.y < vecBoundsMin.y ) vecBoundsMin.y = Min.y;
        if ( Min.z < vecBoundsMin.z ) vecBoundsMin.z = Min.z;
        if ( Max.x > vecBoundsMax.x ) vecBoundsMax.x = Max.x;
        if ( Max.y > vecBoundsMax.y ) vecBoundsMax.y = Max.y;
        if ( Max.z > vecBoundsMax.z ) vecBoundsMax.z = Max.z;

        // Store this in the top detail area list
        AreaList.push_back( pDetailArea );

    } // Next Polygon
```

Finally, with the root node's bounding volume calculated and with PolyList and AreaList containing copies of all the polygon and detail area pointers, we can now step into the recursive BuildTree method starting at the root node and let it build the entire tree from the root node down.

```
    // Build the tree itself
    if (!BuildTree( m_pRootNode, PolyList, AreaList, vecBoundsMin, vecBoundsMax ))
```

```
        return false;
```

At this point the tree will be completely built so we call the CBaseTree::PostBuild method and let it do any final preparation on the tree data. We will see in the next lesson how this method not only calculates and stores the bounding boxes for each polygon stored in the tree (as we have already shown) but also initializes CBaseTree's render system.

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

BuildTree - COctTree

The BuildTree method has the task of dividing the current node's polygon and detail area lists into eight children instead of four and creating eight child nodes. However, with the exception that we must now carve the node into octants using three split planes and creating eight child lists of both polygon and detail area data, the function is essentially the same as the quad-tree version.

Notice in this first section of code how we now instantiate local arrays of eight polygon lists and eight detail area lists and also allocate an array to hold three split planes instead of two.

```
bool COctTree::BuildTree( COctTreeNode * pNode,
                          PolygonList PolyList,
                          DetailAreaList AreaList,
                          const D3DXVECTOR3 & BoundsMin,
                          const D3DXVECTOR3 & BoundsMax )
{
    D3DXVECTOR3                 Normal;
    PolygonList::iterator       PolyIterator;
    DetailAreaList::iterator    AreaIterator;
    CPolygon                  * CurrentPoly, * FrontSplit, * BackSplit;
    PolygonList                 ChildList[8];
    DetailAreaList              ChildAreaList[8];
    CCollision::CLASSIFYTYPE    Location[3];
    D3DXPLANE                   Planes[3];
    unsigned long               i;
    bool                        bStopCode;

    // Store the bounding box properties in the node
    pNode->BoundsMin = BoundsMin;
    pNode->BoundsMax = BoundsMax;

    // Calculate 'Stop' code
    D3DXVECTOR3 vecLeafSize = BoundsMax - BoundsMin;

    bStopCode  = (AreaList.size() == 0 && PolyList.size() == 0) ||
                 (AreaList.size() <= m_nMinAreaCount &&
                  PolyList.size() <= m_nMinPolyCount ) ||
                  D3DXVec3Length( &vecLeafSize ) <= m_fMinLeafSize;
```

Just as before we store the passed bounding box extents in the node and then calculate the leaf size. Notice that this time when calculating the leaf size variable; we also take the y extents into account. Remember, in the quad-tree case we only subtracted the x and z extents of the bounding box to get the diagonal node length. This was because in a quad-tree, we do not partition space vertically. However, in the case of the oct-tree, we take the full diagonal length of the box into account so that the height of a node's bounding box is also a factor in whether or not it becomes a terminal node. As you can see, by simply subtracting the minimum box extents from the maximum box extents, we get a vector whose length describes the minimum size a node can be before it automatically becomes a leaf.

If the stop code variable is true, it means there are two few polygons or detail areas in this node or that the node is too small. Either way, we no longer wish to further partition the space of this node and instead wish to make it a leaf. The following conditional code that makes the node a leaf is completely unchanged from the quad-tree case. It creates a new CBaseLeaf and loops through every polygon and detail area in the node's list and adds them to the leaf. If clipping is being used it is at this point that the pointer of each potentially clipped polygon is also added back to the tree's main polygon list. The node's leaf pointer is then assigned to point at this new leaf and the leaf's bounding box is set to be the same as the node. At this point the node has been successfully turned into a leaf and we return.

```
    // If we reached our stop limit, build a leaf here
    if ( bStopCode )
    {
        // Build a leaf
        CBaseLeaf * pLeaf = new CBaseLeaf( this );
        if ( !pLeaf ) return false;

        // Store the polygons
        for ( PolyIterator = PolyList.begin();
              PolyIterator != PolyList.end(); ++PolyIterator )
        {
            // Retrieve poly
            CurrentPoly = *PolyIterator;
            if ( !CurrentPoly ) continue;

            // Add to full tree polygon list ONLY if splitting was allowed
            if ( m_bAllowSplits ) AddPolygon( CurrentPoly );

            // Also add a reference to the leaf's list
            pLeaf->AddPolygon( CurrentPoly );

        } // Next Polygon

        // Store the area lists
        for ( AreaIterator = AreaList.begin();
              AreaIterator != AreaList.end(); ++AreaIterator )
        {
            // Retrieve detail area item
            TreeDetailArea * pDetailArea = *AreaIterator;
            if ( !pDetailArea ) continue;

            // Add a reference to the leaf's list
            pLeaf->AddDetailArea( pDetailArea );

        } // Next Polygon
```

```
        // Store pointer to leaf in the node and the bounds
        pNode->Leaf = pLeaf;

        // Store the bounding box in the leaf
        pLeaf->SetBoundingBox( pNode->BoundsMin, pNode->BoundsMax );

        // Store the leaf in the leaf list
        AddLeaf( pLeaf );

        // We have reached a leaf
        return true;

    } // End if reached stop code
```

If we make it this far in the function then it means the node is not a leaf and we will further divide its space and its data into eight children. To divide an oct-tree node, we create three planes with which to split and assign the polygon and detail area data to each child list. The first two planes are identical to the planes of the quad-tree node (a plane facing down the positive Z axis and a plane facing down the positive X axis). Now we add a third plane whose normal faces along the positive Y axis and splits space vertically. All three planes intersect at the center of the node and therefore the point on plane used to create each plane is simply the center point of the box.

```
    // Generate the three split planes
    D3DXPlaneFromPointNormal( &Planes[0],
                              &((BoundsMin + BoundsMax) / 2.0f),
                              &D3DXVECTOR3( 0.0f, 0.0f, 1.0f ) );

    D3DXPlaneFromPointNormal( &Planes[1],
                              &((BoundsMin + BoundsMax) / 2.0f),
                              &D3DXVECTOR3( 1.0f, 0.0f, 0.0f ) );

    D3DXPlaneFromPointNormal( &Planes[2],
                              &((BoundsMin + BoundsMax) / 2.0f),
                              &D3DXVECTOR3( 0.0f, 1.0f, 0.0f ) );
```

If we are building a clipped tree then we will need to loop through each of these three planes, and for each plane we will need to classify every polygon in the node's list against it. If we find any polygon spanning a plane, it is split by the plane, the original polygon is deleted from the list, and the two new split polygons are added. This code is exactly the same as the previous versions of the function we have seen with the exception that the outer loop now iterates through three planes instead of two.

```
    // Split all polygons against all three planes if required
    if ( m_bAllowSplits )
    {
        for ( i = 0; i < 3; ++i )
        {
            for ( PolyIterator = PolyList.begin();
                  PolyIterator != PolyList.end(); ++PolyIterator )
            {
                // Store current poly
                CurrentPoly = *PolyIterator;
                if ( !CurrentPoly ) continue;
```

```
        // Classify the poly against the first plane
        Location[0] = CCollision::PolyClassifyPlane(CurrentPoly->m_pVertex,
                                                    CurrentPoly->m_nVertexCount,
                                                    sizeof(CVertex),
                                                    (D3DXVECTOR3&)Planes[i],
                                                    Planes[i].d );

             if ( Location[0] == CCollision::CLASSIFY_SPANNING )
             {
                 // Split the current poly against the plane
                 CurrentPoly->Split( Planes[i], &FrontSplit, &BackSplit );

                 delete CurrentPoly;
                 *PolyIterator = NULL;

                 // Add these to the end of the current poly list
                 PolyList.push_back( FrontSplit );
                 PolyList.push_back( BackSplit );

             } // End if Spanning

         } // Next Polygon

    } // Next Plane

} // End if allow splits
```

At this point if clipping was desired all the polygons in the node's list will have been clipped and will fit neatly into one of the octants of the node. Our task is to loop through every polygon in the list and find out in which octant it is contained. This is done by classifying the polygon against the three planes and analyzing the result. For example, we know that if it is behind the first plane it must lay in the back halfspace of the node. If it is in front of the second plane we know that it is somewhere in the back right area of the node. Finally, if it is in front of the third plane we know that it is contained in the upper back right octant of the node. The strategy in this code is exactly the same as in the quad-tree version except now we have an extra plane to classify each polygon against, adding a few more conditionals.

```
// Classify the polygons and sort them into the child lists.
for ( PolyIterator = PolyList.begin();
      PolyIterator != PolyList.end(); ++PolyIterator )
{
    // Store current poly
    CurrentPoly = *PolyIterator;
    if ( !CurrentPoly ) continue;

    // Classify the poly against the planes
    Location[0] = CCollision::PolyClassifyPlane( CurrentPoly->m_pVertex,
                                                 CurrentPoly->m_nVertexCount,
                                                 sizeof(CVertex),
                                                 (D3DXVECTOR3&)Planes[0],
                                                 Planes[0].d );

    Location[1] = CCollision::PolyClassifyPlane( CurrentPoly->m_pVertex,
                                                 CurrentPoly->m_nVertexCount,
```

```
                                                    sizeof(CVertex),
                                                    (D3DXVECTOR3&)Planes[1],
                                                    Planes[1].d );

        Location[2] = CCollision::PolyClassifyPlane( CurrentPoly->m_pVertex,
                                                    CurrentPoly->m_nVertexCount,
                                                    sizeof(CVertex),
                                                    (D3DXVECTOR3&)Planes[2],
                                                    Planes[2].d );
```

With the classification results for the current polygon stored in the three element Location array, let us now analyze the three results and work out in which octants the polygon is contained and to which child lists the polygon's pointers should be added.

```
        // Position relative to XY plane
        if ( Location[0] == CCollision::CLASSIFY_BEHIND ||
             Location[0] == CCollision::CLASSIFY_SPANNING )
        {
            // Position relative to ZY plane
            if ( Location[1] == CCollision::CLASSIFY_BEHIND ||
                 Location[1] == CCollision::CLASSIFY_SPANNING )
            {
                // Position relative to XZ plane
                if ( Location[2] == CCollision::CLASSIFY_BEHIND ||
                     Location[2] == CCollision::CLASSIFY_SPANNING)
                                    ChildList[0].push_back( CurrentPoly );

                if ( Location[2] == CCollision::CLASSIFY_INFRONT ||
                     Location[2] == CCollision::CLASSIFY_ONPLANE ||
                     Location[2] == CCollision::CLASSIFY_SPANNING)
                                     ChildList[4].push_back( CurrentPoly );

            } // End if behind

            if ( Location[1] == CCollision::CLASSIFY_INFRONT ||
                 Location[1] == CCollision::CLASSIFY_ONPLANE ||
                 Location[1] == CCollision::CLASSIFY_SPANNING )
            {
                // Position relative to XZ plane
                if ( Location[2] == CCollision::CLASSIFY_BEHIND ||
                     Location[2] == CCollision::CLASSIFY_SPANNING)
                                    ChildList[1].push_back( CurrentPoly );

                if ( Location[2] == CCollision::CLASSIFY_INFRONT ||
                     Location[2] == CCollision::CLASSIFY_ONPLANE ||
                     Location[2] == CCollision::CLASSIFY_SPANNING)
                                    ChildList[5].push_back( CurrentPoly );

            } // End if in-front or on-plane

        } // End if behind

        if ( Location[0] == CCollision::CLASSIFY_INFRONT ||
             Location[0] == CCollision::CLASSIFY_ONPLANE ||
             Location[0] == CCollision::CLASSIFY_SPANNING )
```

```
        {
            // Position relative to ZY plane
            if ( Location[1] == CCollision::CLASSIFY_BEHIND ||
                 Location[1] == CCollision::CLASSIFY_SPANNING )
            {
                // Position relative to XZ plane
                if ( Location[2] == CCollision::CLASSIFY_BEHIND ||
                     Location[2] == CCollision::CLASSIFY_SPANNING)
                                    ChildList[2].push_back( CurrentPoly );

                if ( Location[2] == CCollision::CLASSIFY_INFRONT ||
                     Location[2] == CCollision::CLASSIFY_ONPLANE ||
                     Location[2] == CCollision::CLASSIFY_SPANNING)
                                    ChildList[6].push_back( CurrentPoly );

            } // End if behind

            if ( Location[1] == CCollision::CLASSIFY_INFRONT ||
                 Location[1] == CCollision::CLASSIFY_ONPLANE ||
                 Location[1] == CCollision::CLASSIFY_SPANNING )
            {
                // Position relative to XZ plane
                if ( Location[2] == CCollision::CLASSIFY_BEHIND ||
                     Location[2] == CCollision::CLASSIFY_SPANNING)
                                    ChildList[3].push_back( CurrentPoly );

                if ( Location[2] == CCollision::CLASSIFY_INFRONT ||
                     Location[2] == CCollision::CLASSIFY_ONPLANE ||
                     Location[2] == CCollision::CLASSIFY_SPANNING)
                                    ChildList[7].push_back( CurrentPoly );

            } // End if in-front or on-plane

        } // End if in-front or on-plane

    } // Next Triangle
```

At this point we have added all the polygon pointers for this node to the child lists so we must also do the same for the detail areas. The next section of code classifies the detail areas of this node against the three node planes to find out which child lists the detail areas should be added to. The code is almost the same as the quad-tree version with the exception that we now have an extra plane to classify against and more results to analyze.

```
    // Classify the areas and sort them into the child lists.
    for ( AreaIterator = AreaList.begin();
          AreaIterator != AreaList.end(); ++AreaIterator )
    {
        // Store current area
        TreeDetailArea * pDetailArea = *AreaIterator;
        if ( !pDetailArea ) continue;

        // Classify the area against the planes
        Location[0] = CCollision::AABBClassifyPlane( pDetailArea->BoundsMin,
                                                     pDetailArea->BoundsMax,
                                                     (D3DXVECTOR3&)Planes[0],
```

```
                                                Planes[0].d );

    Location[1] = CCollision::AABBClassifyPlane( pDetailArea->BoundsMin,
                                                 pDetailArea->BoundsMax,
                                                 (D3DXVECTOR3&)Planes[1],
                                                 Planes[1].d );

    Location[2] = CCollision::AABBClassifyPlane( pDetailArea->BoundsMin,
                                                 pDetailArea->BoundsMax,
                                                 (D3DXVECTOR3&)Planes[2],
                                                 Planes[2].d );

    // Position relative to XY plane
    if ( Location[0] == CCollision::CLASSIFY_BEHIND ||
         Location[0] == CCollision::CLASSIFY_SPANNING )
    {
        // Position relative to ZY plane
        if ( Location[1] == CCollision::CLASSIFY_BEHIND ||
             Location[1] == CCollision::CLASSIFY_SPANNING )
        {
            // Position relative to XZ plane
            if ( Location[2] == CCollision::CLASSIFY_BEHIND ||
                 Location[2] == CCollision::CLASSIFY_SPANNING)
                                ChildAreaList[0].push_back( pDetailArea );

            if ( Location[2] == CCollision::CLASSIFY_INFRONT ||
                 Location[2] == CCollision::CLASSIFY_SPANNING)
                                ChildAreaList[4].push_back( pDetailArea );

        } // End if behind

        if ( Location[1] == CCollision::CLASSIFY_INFRONT ||
             Location[1] == CCollision::CLASSIFY_SPANNING )
        {
            // Position relative to XZ plane
            if ( Location[2] == CCollision::CLASSIFY_BEHIND ||
                 Location[2] == CCollision::CLASSIFY_SPANNING)
                                ChildAreaList[1].push_back( pDetailArea );

            if ( Location[2] == CCollision::CLASSIFY_INFRONT ||
                 Location[2] == CCollision::CLASSIFY_SPANNING)
                                ChildAreaList[5].push_back( pDetailArea );

        } // End if in-front or on-plane

    } // End if behind


    if ( Location[0] == CCollision::CLASSIFY_INFRONT ||
         Location[0] == CCollision::CLASSIFY_SPANNING )
    {
        // Position relative to ZY plane
        if ( Location[1] == CCollision::CLASSIFY_BEHIND ||
             Location[1] == CCollision::CLASSIFY_SPANNING )
        {
            // Position relative to XZ plane
            if ( Location[2] == CCollision::CLASSIFY_BEHIND ||
```

```
                    Location[2] == CCollision::CLASSIFY_SPANNING)
                                   ChildAreaList[2].push_back( pDetailArea );

               if ( Location[2] == CCollision::CLASSIFY_INFRONT ||
                    Location[2] == CCollision::CLASSIFY_SPANNING)
                                   ChildAreaList[6].push_back( pDetailArea );

           } // End if behind

           if ( Location[1] == CCollision::CLASSIFY_INFRONT ||
                Location[1] == CCollision::CLASSIFY_SPANNING )
           {
               // Position relative to XZ plane
               if ( Location[2] == CCollision::CLASSIFY_BEHIND ||
                    Location[2] == CCollision::CLASSIFY_SPANNING)
                                   ChildAreaList[3].push_back( pDetailArea );

               if ( Location[2] == CCollision::CLASSIFY_INFRONT ||
                    Location[2] == CCollision::CLASSIFY_SPANNING)
                                   ChildAreaList[7].push_back( pDetailArea );

           } // End if in-front or on-plane

       } // End if in-front or on-plane

   } // Next Detail Area
```

At this point we have the eight polygon lists and the eight detail area lists, so it is time to create each child node, compute its bounding box, and store a pointer to each child node in the parent node's child list.

```
   // Build each of the children here
   for( i = 0; i < 8; ++i )
   {
       // Calculate child bounding box values
       D3DXVECTOR3 NewBoundsMin, NewBoundsMax,
                   MidPoint = (BoundsMin + BoundsMax) / 2.0f;
       switch( i )
       {
           case 0: // Bottom Behind Left
               NewBoundsMin = BoundsMin;
               NewBoundsMax = D3DXVECTOR3( MidPoint.x, MidPoint.y, MidPoint.z);
               break;

           case 1: // Bottom Behind Right
               NewBoundsMin = D3DXVECTOR3( MidPoint.x, BoundsMin.y, BoundsMin.z );
               NewBoundsMax = D3DXVECTOR3( BoundsMax.x, MidPoint.y, MidPoint.z);
               break;

           case 2: // Bottom InFront Left
               NewBoundsMin = D3DXVECTOR3( BoundsMin.x, BoundsMin.y, MidPoint.z );
               NewBoundsMax = D3DXVECTOR3( MidPoint.x, MidPoint.y, BoundsMax.z );
               break;

           case 3: // Bottom InFront Right
```

```
                NewBoundsMin = D3DXVECTOR3( MidPoint.x, BoundsMin.y, MidPoint.z );
                NewBoundsMax = D3DXVECTOR3( BoundsMax.x, MidPoint.y, BoundsMax.z );
                break;

            case 4: // Top Behind Left
                NewBoundsMin = D3DXVECTOR3( BoundsMin.x, MidPoint.y, BoundsMin.z );
                NewBoundsMax = D3DXVECTOR3( MidPoint.x, BoundsMax.y, MidPoint.z);
                break;

            case 5: // Top Behind Right
                NewBoundsMin = D3DXVECTOR3( MidPoint.x, MidPoint.y, BoundsMin.z );
                NewBoundsMax = D3DXVECTOR3( BoundsMax.x, BoundsMax.y, MidPoint.z);
                break;

            case 6: // Top InFront Left
                NewBoundsMin = D3DXVECTOR3( BoundsMin.x, MidPoint.y, MidPoint.z );
                NewBoundsMax = D3DXVECTOR3( MidPoint.x, BoundsMax.y, BoundsMax.z );
                break;

            case 7: // Top InFront Right
                NewBoundsMin = D3DXVECTOR3( MidPoint.x, MidPoint.y, MidPoint.z );
                NewBoundsMax = BoundsMax;
                break;

        } // End Child Type Switch

        // Allocate child node
        pNode->Children[i] = new COctTreeNode;
        if ( !pNode->Children[i] ) return false;

        // Recurse into this new node
        BuildTree( pNode->Children[i],
                   ChildList[i],
                   ChildAreaList[i],
                   NewBoundsMin,
                   NewBoundsMax );

        // Clean up
        ChildList[i].clear();
        ChildAreaList[i].clear();

    } // Next Child

    // Success!
    return true;
}
```

At the end of the child creation loop, once the node has been attached to the parent, we traverse into that node by passing its pointer, its polygon and detail area lists, and its bounding box into another recursion of the BuildTree method.

We have now covered all the code to build an oct-tree. It should be clear to you that having the class hierarchy and the housekeeping methods tucked away in CBaseTree and CBaseLeaf, makes creating new tree types simple and requires very little code.

14.19.3 The Oct-Tree Query Methods

The methods to query the oct-tree are almost identical to their quad-tree counterparts, so we will cover them only briefly. Each function works in exactly the same way with the exception that eight children will need to be tested and traversed into instead of four.

CollectLeavesAABB - COctTree

The CollectLeavesAABB method is actually identical to its quad-tree counterpart. It is passed an empty leaf list and a bounding box describing the query volume. The function simply wraps a call to the CollectAABBRecurse method for the root node.

```
bool COctTree::CollectLeavesAABB( LeafList & List,
                                  const D3DXVECTOR3 & Min,
                                  const D3DXVECTOR3 & Max )
{
    // Trigger Recursion
    return CollectAABBRecurse( m_pRootNode, List, Min, Max );
}
```

CollectAABBRecurse – COctTree

This recursive method steps through every leaf in the tree that is contained or partially contained in the passed query volume. For a leaf found inside the query volume, its pointer is added to the passed leaf list so it can be returned to the application. If it is determined at any node that its volume does not intersect the query volume we can refrain from further traversing that branch of the tree. Otherwise, we must traverse into any child nodes that do intersect the query volume. As soon as a leaf is found, its pointer is added to the passed leaf list. If at any point the query volume is found to completely contain the node volume, we know that all the children of that node must also be contained. In such a situation the bAutoCollect boolean is set to true and for every child node under the contained node, we no longer perform AABB tests and automatically step into each of its children until the leaf nodes are encountered.

```
bool COctTree::CollectAABBRecurse( COctTreeNode * pNode,
                                   LeafList & List,
                                   const D3DXVECTOR3 & Min,
                                   const D3DXVECTOR3 & Max,
                                   bool bAutoCollect /* = false */ )
{
    bool  bResult = false;
    ULONG i;

    // Validate parameters
    if ( !pNode ) return false;

    // Does the specified box intersect this node?
```

```
    if ( !bAutoCollect && !CCollision::AABBIntersectAABB( bAutoCollect,
                                                          Min,
                                                          Max,
                                                          pNode->BoundsMin,
                                                          pNode->BoundsMax ) )
                                                          return false;

    // Is there a leaf here, add it to the list
    if ( pNode->Leaf ) { List.push_back( pNode->Leaf ); return true; }

    // Traverse down to children
    for ( i = 0; i < 8; ++i )
    {
        if ( CollectAABBRecurse( pNode->Children[i],
                                 List,
                                 Min,
                                 Max,
                                 bAutoCollect ) ) bResult = true;

    } // Next Child

    // Return the 'was anything added' result.
    return bResult;
}
```

So the only difference between this method and its quad-tree counterpart is the fact that it loops through eight children instead of four. You should be able to see a pattern forming here that will allow you to create any tree type you desire by implementing only a few core functions.

CollectLeavesRay - COctTree

The CollectLeavesRay method is called by the application to query the oct-tree for any leaves that intersect the passed ray. This method is simply a wrapper around a call to the CollectRayRecurse method for the root node. It is unchanged from its quad-tree counterpart.

```
bool COctTree::CollectLeavesRay( LeafList & List,
                                 const D3DXVECTOR3 & RayOrigin,
                                 const D3DXVECTOR3 & Velocity )
{
    return CollectRayRecurse( m_pRootNode, List, RayOrigin, Velocity );
}
```

CollectRayRecurse - COctTree

This method is almost unchanged from its quad-tree counterpart with the exception that it steps into eight children at each non-terminal node instead of four.

```
bool COctTree::CollectRayRecurse( COctTreeNode * pNode,
                                  LeafList & List,
                                  const D3DXVECTOR3 & RayOrigin,
                                  const D3DXVECTOR3 & Velocity )
{
    bool  bResult = false;
    ULONG i;
    float t;

    // Validate parameters
    if ( !pNode ) return false;

    // Does the ray intersect this node?
    if ( !CCollision::RayIntersectAABB( RayOrigin,
                                        Velocity,
                                        pNode->BoundsMin,
                                        pNode->BoundsMax,
                                        t ) ) return false;


    // Is there a leaf here, add it to the list
    if ( pNode->Leaf ) { List.push_back( pNode->Leaf ); return true; }


    // Traverse down to children
    for ( i = 0; i < 8; ++i )
    {
        if ( CollectRayRecurse( pNode->Children[i],
                                List,
                                RayOrigin,
                                Velocity ) ) bResult = true;

    } // Next Child

    // Return the 'was anything added' result.
    return bResult;
}
```

And there we have it. We have implemented the query methods for the oct-tree with only a few changes to the code for the quad-tree.

14.19.4 The Oct-Tree Debug Draw Routine

As you have seen, most of the oct-tree functions were essentially the same as their quad-tree counterparts with the exception that eight children have to be processed at each node instead of four.

Because of this, we will not spend time showing the code to the COctTree::DebugDraw method even though it has been implemented in the source code. If you examine the code you will see that it is almost identical to its quad-tree counterpart with the exception that it steps into eight children.

This concludes our discussion and implementation of oct-trees for this lesson. We now have three tree types (quad-tree, YV quad-tree, oct-tree) to choose from for use as a spatial manager in our future applications. We have just one more tree type to create in this lesson: the kD-tree.

14.20 kD-Tree Implementation

The kD-tree is really the simplest tree to implement due to the fact that we only have a single plane used to partition space at each node. Implementing the kD-tree will once again involve deriving a class from CBaseTree and implementing the core functions to build and query the tree. The kD-tree class declaration and implementation can be found in the project source files CKDTree.h and CKDTree.cpp. As with all tree types, we will implement a node type specific to the tree type.

Although the kD-tree object's query function could be implemented in exactly the same way as the quad-tree and oct-tree versions (only with two children instead of one) we will use the CKDTree::CollectLeavesRay method to implement a traversal strategy closer to what we will see when we introduce the BSP tree. You will recall that while, technically speaking, a kD-tree is a BSP tree because it partitions space into two halfspaces at each node, the kD-tree carves the world into axis aligned bounding boxes that fit neatly into each leaf node. Therefore, with a kD-tree, just like an oct-tree and a quad-tree, the query methods that traverse the tree can locate the leaves that a ray intersects by performing ray/box intersection tests if desired. However, there is an alternative that we will explore that does not use box testing.

A BSP tree (see Chapters 16 and 17) is a binary tree that partitions space into two halfspaces at each node just like the kD-tree. However, the split planes of a BSP tree do not have to be axis aligned; in fact they are usually based on the polygon data being compiled. While this will all make a lot more sense later in the course (and beyond, when we really get into using BSP trees), just know for now that the leaves of a BSP tree will not all be box shaped. The leaves can be arbitrarily shaped convex hulls which cannot be neatly represented with an AABB. Therefore, we cannot actually traverse a BSP tree in quite the same way when we want to find the leaves. The CollectLeavesRay method of the kD-tree will be a more BSP-centric version that traverses the tree performing a series of ray/plane tests instead of ray/box tests. This will give us some early insight into the BSP traversals we will be performing in later chapters. However, it is worth noting that in the case of the kD-tree, where the leaves are all box shaped, you could replace our CollectLeavesRay method with a version that is almost identical to the ones we have seen for other tree types.

14.20.1 CKDTreeNode – The Source Code

Each node in our KD-tree will be an object of type CKDTreeNode which is implemented in the project source files CKDTree.h and CKDTree.cpp. Just like the other node types we have seen it stores a bounding box and a pointer to a CBaseLeaf object which will be used only by terminal nodes. Because a kD-tree only has two children, we will not store the child pointers in an array but will instead simply have two pointers called Front and Back. The names of these children obviously describe its position relative to the parent node's single split plane. Like all other node types, we implement the recursive SetVisible method so that the tree can inform a node to start a recursive process that will flag all child leaves of the node as either visible or invisible. Finally, notice in the following class declaration that unlike the quad-tree and oct-tree implementations, where the node split planes were created temporarily and then discarded, in the case of the kD-tree, we have decided to store the single split plane used to partition the node's space during the build process.

```
class CKDTreeNode
{
public:

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

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

    // Public Variables for This Class
    D3DXPLANE       Plane;              // Splitting plane for this node
    CKDTreeNode *   Front;              // Node in front of the plane
    CKDTreeNode *   Back;               // Node behind the plane
    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 'last plane' index.
};
```

The reason we have decided to store the node plane in the kD-tree case is because this is more in keeping with the traditional BSP approach. In a BSP tree, each node stores a single split plane and pointers to two children that represent the halfspaces of that plane. By storing the plane in the kD-tree node, we enable our kD-tree to have the ability to work more like a traditional BSP tree if we so desire (again, in many respects they are basically the same type of tree, so this works quite well).

For example, in the CollectLeavesAABB method we will implement it using the same style as the quad-tree and oct-tree. The tree will be traversed with a query volume that will be tested for intersection against the bounding box of each child. However, in the CollectLeavesRay method we will implement the method just as we would for a BSP tree. We will assume that we have no AABB stored at each node; only a split plane. This method will send the ray down the tree and at each node test to see if the ray is spanning the node. If it is, the ray is split in two and each half of the ray is sent into the respective child for the halfspace the ray segment is contained within. Eventually, all of our split ray fragments will pop out in leaf nodes. At that point, the leaves are added to the collection list (more on this later). In order for

such a method to be implemented we need to have a tree where each node represents a split plane, just like a BSP tree. By storing the split plane in the kD-tree node and implementing our ray query method as a BSP method, we draw attention to the similarities between the kD-tree and the BSP tree. This should prove to be helpful when we move on to examine BSP trees.

SetVisible - CKDTreeNode

The SetVisible method of the kD-tree's node object is the simplest we have seen so far. If the current node being visited points to a leaf object, then this is a terminal node and we set the visibility status of the attached leaf and return. Otherwise, we are in a normal kD-tree node and we must traverse into both children.

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

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

This function is simpler than the oct-tree and quad-tree case because we have only two children to process and no need to loop.

14.20.2 The CKDTree Source Code

The CKDTree class looks almost identical to the quad-tree and oct-tree class declarations due to the fact that it is derived from CBaseTree and has to implement all the same building and querying methods.

```
class CKDTree : public CBaseTree
{
public:

    // Constructors & Destructors for This Class.
    virtual ~CKDTree();
             CKDTree( LPDIRECT3DDEVICE9 pDevice,
                      bool  bHardwareTnL,
                      float fMinLeafSize = 300.0f,
                      ULONG nMinPolyCount = 600,
                      ULONG nMinAreaCount = 0 );

    // Public Virtual Functions for This Class (from base).
    virtual bool Build                 ( bool bAllowSplits = true );

    virtual void ProcessVisibility     ( CCamera & Camera );
```

```
    virtual bool CollectLeavesAABB      ( LeafList & List,
                                          const D3DXVECTOR3 & Min,
                                          const D3DXVECTOR3 & Max );

    virtual bool CollectLeavesRay       ( LeafList & List,
                                          const D3DXVECTOR3 & RayOrigin,
                                          const D3DXVECTOR3 & Velocity );

    virtual void DebugDraw              ( CCamera & Camera );

    virtual bool GetSceneBounds         ( D3DXVECTOR3 & Min, D3DXVECTOR3 & Max );

protected:

    // Protected virtual Functions for This Class
    virtual bool BuildTree              ( CKDTreeNode * pNode,
                                          PolygonList PolyList,
                                          DetailAreaList AreaList,
                                          const D3DXVECTOR3 & BoundsMin,
                                          const D3DXVECTOR3 & BoundsMax,
                                          ULONG PlaneType = 0 );


    // Protected Functions for This Class
    void         UpdateTreeVisibility   ( CKDTreeNode * pNode,
                                          CCamera & Camera,
                                          UCHAR FrustumBits = 0x0 );

    bool         DebugDrawRecurse       ( CKDTreeNode * pNode,
                                          CCamera & Camera,
                                          bool bRenderInLeaf );

    bool         CollectAABBRecurse     ( CKDTreeNode * pNode,
                                          LeafList & List,
                                          const D3DXVECTOR3 & Min,
                                          const D3DXVECTOR3 & Max,
                                          bool bAutoCollect = false );

    bool         CollectRayRecurse      ( CKDTreeNode * pNode,
                                          LeafList & List,
                                          const D3DXVECTOR3 & RayOrigin,
                                          const D3DXVECTOR3 & Velocity );


    // Protected Variables for This Class

    CKDTreeNode * m_pRootNode;          // The root node of the tree
    bool          m_bAllowSplits;       // Is splitting allowed?
    float         m_fMinLeafSize;       // Min leaf size stop code
    ULONG         m_nMinPolyCount;      // Min polygon count stop code
    ULONG         m_nMinAreaCount;      // Min detail area count stop code

};
```

As you can see, the only difference in the member variable list is that it now stores a root node pointer of type CKDTreeNode.

Build - CKDTree

We now know that the Build function of our derived classes is responsible for allocating the root node and calculating its bounding box. It then begins the recursive building process, starting at the root, with a call to the BuildTree method.

The Build method of the kD-tree is almost identical to the others we have seen, so we will cover it only briefly here.

The function first allocates a new CKDTreeNode object and assigns the tree's root node pointer to point at it. We then store that passed boolean parameter so that we know during the building process whether or not we are creating a clipped tree.

```
bool CKDTree::Build( bool bAllowSplits /* = true */ )
{
    PolygonList::iterator       PolyIterator = m_Polygons.begin();
    DetailAreaList::iterator    AreaIterator = m_DetailAreas.begin();
    PolygonList                 PolyList;
    DetailAreaList              AreaList;
    unsigned long               i;

    // Reset our tree info values.
    D3DXVECTOR3 vecBoundsMin( FLT_MAX, FLT_MAX, FLT_MAX );
    D3DXVECTOR3 vecBoundsMax( -FLT_MAX, -FLT_MAX, -FLT_MAX );

    // Allocate a new root node
    m_pRootNode = new CKDTreeNode;
    if ( !m_pRootNode ) return false;

    // Store the allow splits value for later retrieval.
    m_bAllowSplits = bAllowSplits;
```

We now need to calculate the bounding box for the root node so we loop through every vertex of every polygon registered with the tree and expand the box to fit all vertices. As we process each polygon, its pointer is also copied into the local polygon list (PolyList). After we have resized the box for every polygon and made a copy of the original tree's polygon list, we test to see if a clipped tree is being built. If so, we empty the tree's polygon list since it will be populated with polygon fragments during clipping.

```
    // Loop through all of the initial polygons
    for ( ; PolyIterator != m_Polygons.end(); ++PolyIterator )
    {
        // Retrieve the polygon
        CPolygon * pPoly = *PolyIterator;
        if ( !pPoly ) continue;

        // Calculate total scene bounding box.
        for ( i = 0; i < pPoly->m_nVertexCount; ++i )
        {
            // Store info
            CVertex * pVertex = &pPoly->m_pVertex[i];
            if ( pVertex->x < vecBoundsMin.x ) vecBoundsMin.x = pVertex->x;
```

```
            if ( pVertex->y < vecBoundsMin.y ) vecBoundsMin.y = pVertex->y;
            if ( pVertex->z < vecBoundsMin.z ) vecBoundsMin.z = pVertex->z;
            if ( pVertex->x > vecBoundsMax.x ) vecBoundsMax.x = pVertex->x;
            if ( pVertex->y > vecBoundsMax.y ) vecBoundsMax.y = pVertex->y;
            if ( pVertex->z > vecBoundsMax.z ) vecBoundsMax.z = pVertex->z;
        }

        // Store this polygon in the top polygon list
        PolyList.push_back( pPoly );

    } // Next Polygon

    // Clear the initial polygon list if we are going to split the polygons
    // as this will eventually become storage for whatever gets built
    if ( bAllowSplits ) m_Polygons.clear();
```

The bounding box is large enough to contain all registered polygons, but we also need to factor in any registered detail areas.

```
    // Loop through all of the detail areas
    for ( ; AreaIterator != m_DetailAreas.end(); ++AreaIterator )
    {
        // Retrieve the detail area
        TreeDetailArea * pDetailArea = *AreaIterator;
        if ( !pDetailArea ) continue;

        // Calculate total scene bounding box.
        D3DXVECTOR3 & Min = pDetailArea->BoundsMin;
        D3DXVECTOR3 & Max = pDetailArea->BoundsMax;
        if ( Min.x < vecBoundsMin.x ) vecBoundsMin.x = Min.x;
        if ( Min.y < vecBoundsMin.y ) vecBoundsMin.y = Min.y;
        if ( Min.z < vecBoundsMin.z ) vecBoundsMin.z = Min.z;
        if ( Max.x > vecBoundsMax.x ) vecBoundsMax.x = Max.x;
        if ( Max.y > vecBoundsMax.y ) vecBoundsMax.y = Max.y;
        if ( Max.z > vecBoundsMax.z ) vecBoundsMax.z = Max.z;

        // Store this in the top detail area list
        AreaList.push_back( pDetailArea );

    } // Next Polygon
```

At this point we call the BuildTree method to start the tree building process from the root node. The function is passed the root node pointer and a list of polygons and detail areas that are contained in the root node's bounding box. The final two parameters represent the AABB that bounds all data contained in the root node.

```
    // Build the tree itself
    if ( !BuildTree( m_pRootNode, PolyList, AreaList, vecBoundsMin,vecBoundsMax))
         return false;

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

When the BuildTree method returns and the entire tree has been compiled, we remember to call the CBaseTree::PostBuild method to give the base class a chance to initialize the rendering system and calculate the AABBs for each polygon in the tree.

BuildTree - CKDTree

This function is a little different from the others we have seen so far. Not only does it split the node using a single plane and compile two lists of polygon and detail area data (one for each child), the plane used to split the node is actually switched between iterations. That is, at the first node we split the node using a plane whose normal faces down the world Z axis. When we step into each of the children, we flip the plane so that it is now aligned with the world X axis. For each of its children the plane is flipped again so that its normal is facing along the world Y axis. Therefore, although at each node we are partitioning space into only two children, the scene is carved up in a very similar way to an oct-tree. As we step through the levels of the kD-tree, the plane is continuously alternated between the three world axis aligned planes, wrapping around to the first plane used after every three levels of depth.

> **Note:** You can also use a kD-tree to partition 2D space. Simply transition between the two planes you are interested in rather than three planes.

The Build function is actually a bit simpler than the others we have seen, although at first glance it may not appear so because it is a little larger. The only reason why this is the case is because we are not generating the child nodes in a loop since there are only two children.

The kD-tree BuildTree method accepts an additional plane flag parameter which defaults to zero the first time it is called for the root node. Every time this method calls itself recursively, this parameter will be set in the range [0, 2] describing which of the three world aligned planes should be used to partition the node. The first time it is called from the Build method no value is passed, which means it is set to zero for the root node. This tells the function that the root node should be divided using the first of the three split planes; the split plane whose normal is pointing down the world z axis. Every time this method calls itself to traverse into children, it will increment this value (wrapping around to zero once it is greater than 2) such that the next world aligned plane will be used for its children. For example, the root node will use plane 0 which is the plane aligned with the world Z axis. Its children will be passed a plane value of 1 describing that they should be partitioned using the plane aligned with the world X axis, and so on.

The first section of the code stores the passed bounding box in the passed node and calculates the size of the node's bounding box. It then uses this box and the number of polygons and detail areas passed into the node to determine if this node should become a leaf.

```
bool CKDTree::BuildTree(      CKDTreeNode * pNode,
                              PolygonList PolyList,
                              DetailAreaList AreaList,
                              const D3DXVECTOR3 & BoundsMin,
                              const D3DXVECTOR3 & BoundsMax,
                              ULONG PlaneType /* = 0 */ )
{
```

```
    D3DXVECTOR3                 Normal;
    PolygonList::iterator       PolyIterator;
    DetailAreaList::iterator    AreaIterator;
    CPolygon                  * CurrentPoly, * FrontSplit, * BackSplit;
    PolygonList                 FrontList, BackList;
    DetailAreaList              FrontAreaList, BackAreaList;
    bool                        bStopCode;

    // Store the bounding box properties in the node
    pNode->BoundsMin = BoundsMin;
    pNode->BoundsMax = BoundsMax;

    // Calculate 'Stop' code
    D3DXVECTOR3 vecLeafSize = BoundsMax - BoundsMin;
    bStopCode  = (AreaList.size() == 0 && PolyList.size() == 0) ||
                 (AreaList.size() <= m_nMinAreaCount &&
                  PolyList.size() <= m_nMinPolyCount) ||
                 D3DXVec3Length( &vecLeafSize ) <= m_fMinLeafSize;
```

As with all other tree types, if the stop code boolean gets set to true, we know we have reached a terminal node. The code for creating the leaf is exactly the same as all other tree types. We create a new CBaseLeaf object and then add every polygon and detail area that made it into this node to the leaf. If a clipped tree is being built, then we also re-add the polygon data that made it into this node to the tree's main polygon list. We then attach the leaf to the node, store the node's bounding box in the leaf, and add the leaf to the tree's leaf list before returning.

```
    // If we reached our stop limit, build a leaf here
    if ( bStopCode )
    {
        // Build a leaf
        CBaseLeaf * pLeaf = new CBaseLeaf( this );
        if ( !pLeaf ) return false;

        // Store the polygons
        for (     PolyIterator = PolyList.begin();
                  PolyIterator != PolyList.end(); ++PolyIterator )
        {
            // Retrieve poly
            CurrentPoly = *PolyIterator;
            if ( !CurrentPoly ) continue;

            // Add to full tree polygon list ONLY if splitting was allowed
            if ( m_bAllowSplits ) AddPolygon( CurrentPoly );

            // Also add a reference to the leaf's list
            pLeaf->AddPolygon( CurrentPoly );

        } // Next Polygon

        // Store the area lists
        for ( AreaIterator = AreaList.begin();
              AreaIterator != AreaList.end(); ++AreaIterator )
        {
            // Retrieve detail area item
            TreeDetailArea * pDetailArea = *AreaIterator;
```

```
            if ( !pDetailArea ) continue;

            // Add a reference to the leaf's list
            pLeaf->AddDetailArea( pDetailArea );

        } // Next Polygon

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

        // Store the bounding box in the leaf
        pLeaf->SetBoundingBox( pNode->BoundsMin, pNode->BoundsMax );

        // Store the leaf in the leaf list
        AddLeaf( pLeaf );

        // We have reached a leaf
        return true;

    } // End if reached stop code
```

If we reach this point in the function, we know we are not at a leaf node and we will have to further partition this node into two halfspaces using the split plane. Which of the three axis aligned split planes we use to create the split plane is determined by the PlaneType parameter passed in. Notice in the next code block that this time we actually store the split plane in the node's Plane member.

```
    // Otherwise we must continue to refine. Choose a plane.
    if ( PlaneType == 0 )
        Normal = D3DXVECTOR3( 0.0f, 0.0f, 1.0f );
    else if ( PlaneType == 1 )
        Normal = D3DXVECTOR3( 1.0f, 0.0f, 0.0f );
    else if ( PlaneType == 2 )
        Normal = D3DXVECTOR3( 0.0f, 1.0f, 0.0f );

    // Generate the actual plane
    D3DXPlaneFromPointNormal( &pNode->Plane,
                              &((BoundsMin + BoundsMax) / 2.0f),
                              &Normal );
```

Now that we have determined which split plane to use at this node, we will loop through every polygon in the node's polygon list and classify it against the plane. If it is in front of the plane we will add it to the front polygon list (the list that will be passed to the front child) and if it is behind the plane we will add it to the back list. In the case where the polygon is on the plane, it does not really matter which child we assign it to (we will add it to the front list).

```
    // Classify all polygons
    for (   PolyIterator = PolyList.begin();
            PolyIterator != PolyList.end(); ++PolyIterator )
    {
        // Store current poly
        CurrentPoly = *PolyIterator;

        // Classify the poly against the plane
```

```
        CCollision::CLASSIFYTYPE Location = CCollision::PolyClassifyPlane
                                                    (CurrentPoly->m_pVertex,
                                                     CurrentPoly->m_nVertexCount,
                                                     sizeof(CVertex),
                                                     (D3DXVECTOR3&)pNode->Plane,
                                                     pNode->Plane.d );

        // Decide what to do
        switch ( Location )
        {
            case CCollision::CLASSIFY_BEHIND:

                // Add straight to the back list
                BackList.push_back( CurrentPoly );
                break;

            case CCollision::CLASSIFY_ONPLANE:
            case CCollision::CLASSIFY_INFRONT:

                // Add straight to the front list
                FrontList.push_back( CurrentPoly );
                break;
```

If the polygon is spanning the plane and we are building a clipped tree, we will split the polygon into two new child polygons and delete the original polygon. We will then add the front split polygon to the front list and the back split polygon to the back list. If the polygon is spanning but we are not building a clipped tree, we will simply assign the polygon to the front and back lists to that it gets assigned to both child nodes in which it is partially contained.

```
            case CCollision::CLASSIFY_SPANNING:

                // Is splitting allowed?
                if ( m_bAllowSplits )
                {
                    // Split the current poly against the plane and delete it
                    CurrentPoly->Split( pNode->Plane, &FrontSplit, &BackSplit );
                    delete CurrentPoly;

                    // Add the fragments (if any survived) to the appropriate lists
                    if ( BackSplit  ) BackList.push_back( BackSplit );
                    if ( FrontSplit ) FrontList.push_back( FrontSplit );
                }
                else
                {
                    // Add to both lists!
                    FrontList.push_back( CurrentPoly );
                    BackList.push_back( CurrentPoly );
                }
                break;
        }
    }
```

With the polygon lists compiled for both children, we will classify the detail areas against the plane and add them to either the front or back list (or both). If the detail area is behind the plane it is added to the

back list of detail objects and if it is in front of the plane it is added to the front detail list. If the detail area's AABB is spanning the plane, it is added to the detail area list of both children.

```
    // Classify all detail areas
    for ( AreaIterator = AreaList.begin();
          AreaIterator != AreaList.end();
        ++AreaIterator )
    {
        // Store current area
        TreeDetailArea * pDetailArea = *AreaIterator;

        // Classify the area against the plane
        CCollision::CLASSIFYTYPE Location = CCollision::AABBClassifyPlane
                                                    (pDetailArea->BoundsMin,
                                                     pDetailArea->BoundsMax,
                                                     (D3DXVECTOR3&)pNode->Plane,
                                                     pNode->Plane.d );

        // Decide what to do
        switch ( Location )
        {
            case CCollision::CLASSIFY_BEHIND:

                // Add straight to the back list
                BackAreaList.push_back( pDetailArea );
                break;

            case CCollision::CLASSIFY_INFRONT:

                // Add straight to the front list
                FrontAreaList.push_back( pDetailArea );
                break;

            case CCollision::CLASSIFY_SPANNING:

                // Add to both the front and back lists
                BackAreaList.push_back( pDetailArea );
                FrontAreaList.push_back( pDetailArea );
                break;

        } // End Switch

    } // Next Detail Area
```

At this point we have the polygon lists and the detail area lists compiled for each child node that we are about to create. Unlike the same method from previous tree types, we will not perform child creation in a loop (we will unroll that loop). This next section of code builds the front child first. It calculates its bounding volume based on the split plane being used by the parent node and then allocates a new CKDTreeNode object which is assigned to the front child pointer of the parent node. We then recur into that node by calling the BuildTree function again for the front child.

```
    D3DXVECTOR3 NewBoundsMin, NewBoundsMax, MidPoint;

    // Calculate box midpoint
```

```
    MidPoint = (BoundsMin + BoundsMax) / 2.0f;

    // Calculate child bounding box values
    NewBoundsMax = BoundsMax;

    if ( PlaneType == 0 ) // XY Plane
        NewBoundsMin = D3DXVECTOR3( BoundsMin.x, BoundsMin.y, MidPoint.z);
    else if ( PlaneType == 1 ) // XZ Plane
        NewBoundsMin = D3DXVECTOR3( MidPoint.x, BoundsMin.y, BoundsMin.z);
    else if ( PlaneType == 2 ) // YZ Plane
        NewBoundsMin = D3DXVECTOR3( BoundsMin.x, MidPoint.y, BoundsMin.z);

    // Allocate child node
    pNode->Front = new CKDTreeNode;
    if ( !pNode->Front ) return false;

    // Recurse into this new node
    BuildTree( pNode->Front,
               FrontList,
               FrontAreaList,
               NewBoundsMin,
               NewBoundsMax,
              (PlaneType + 1) % 3 );

    // Clean up
    FrontList.clear();
    FrontAreaList.clear();
```

As you can see, the above code is calculating the bounding box for the child that is in the front half space of the node. This basically means the child node's bounding volume will be one half of the parent node's volume. If the parent node split its space using the XY plane, the minimum extent of the bounding box will be minx, miny and minz as this describes the minimum extents of a box that starts halfway along the parent node's Z axis (see Figure 14.81). In the case of the YZ plane, the front child's bounding box will be the right half of the parent node, where the minimum X extent will be in the center of the box. In each case, because of the way the normals of the planes are facing, the maximum extents of the front child will always be the maximum extents of the parent node (see Figures 14.81 through 14.83). Figure 14.83 similarly shows how the minimum and maximum extents of the front child's AABB would be calculated using the parent node's bounding box if the parent node partitioned space using the XZ plane.

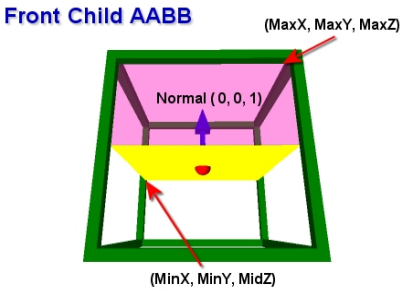


Figure 14.81
Front child of the XY plane.

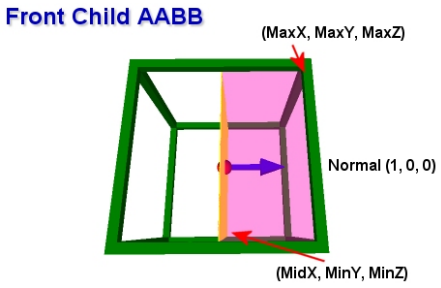


Figure 14.82
Front child of the YZ plane.

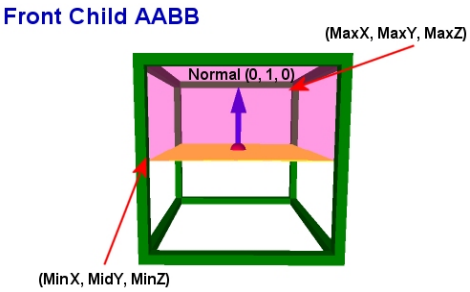


Figure 14.83
Front Child of the XZ Plane.

After we determine which plane was used in the above code we calculate the bounding box for the front child and allocate the new node. The node is then attached to the parent node's front pointer. The BuildTree function then calls itself to step into the front child node and continue the build process down that branch of the tree. Notice that when we call the BuildTree function, we increment the current value of the PlaneType parameter so that the child node will use the next plane in the list of three to partition its space. Also note that we mod this with the number 3 so that as soon as we increment PlaneCount beyond a value of 2, it wraps around to 0 again to start the plane selection process from the beginning. That is, in the child node, the XY plane will be chosen as the split plane.

When the BuildTree method returns above, the front child and all its child nodes and leaves will have been created. Now it is time to build the back child (i.e., the child that is positioned in the back halfspace of the parent node's split plane).

The first thing we do is calculate the bounding box extents of the back child based on the plane that was used as a splitter for the parent node. Because this child is on the back of the parent node's split plane, the bounding box extents of the child will range from min to midpoint on the axis aligned with the plane. The minimum extents of the back child will always be equal to the minimum extents of the parent node's volume for all components, regardless of the split plane being used.

```
    // Calculate child bounding box values
    NewBoundsMin = BoundsMin;

    if ( PlaneType == 0 ) // XY Plane
        NewBoundsMax = D3DXVECTOR3( BoundsMax.x, BoundsMax.y, MidPoint.z);
    else if ( PlaneType == 1 ) // XZ Plane
        NewBoundsMax = D3DXVECTOR3( MidPoint.x, BoundsMax.y, BoundsMax.z);
    else if ( PlaneType == 2 ) // YZ Plane
        NewBoundsMax = D3DXVECTOR3( BoundsMax.x, MidPoint.y, BoundsMax.z);
```

If you are having trouble picturing what we are doing then take a look at Figures 14.84 through 14.86. The figures depict how the bounding box of the back child is calculated using the components of the parent node's volume based on chosen split plane.

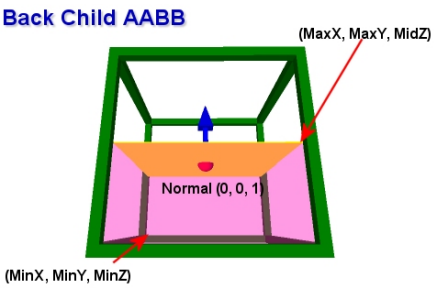


Figure 14.84
Back child of the XY plane.

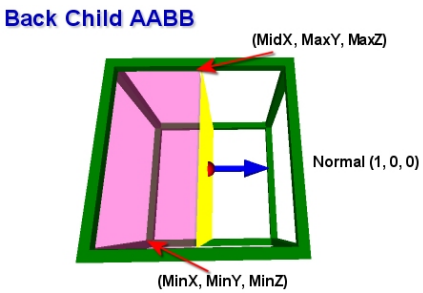


Figure 14.85
Back child of the YZ plane.

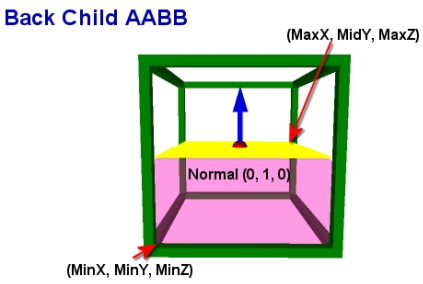


Figure 14.86
Back child of the XZ plane

With the bounding box calculated for the back child we allocate a new CKDTreeNode and attach it to the parent node's Back child pointer. We then call the BuildTree method and recur into the back child passing in the polygon and detail area lists for that node along with its bounding volume. Once again, we increment the PlaneType value that was passed into this instance of the function so that in the next level of the tree we use the next split plane. The modulus is performed so that we wrap this value around to 0 should it exceed 2.

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

    // Recurse into this new node
    BuildTree( pNode->Back,
               BackList,
               BackAreaList,
               NewBoundsMin,
               NewBoundsMax,
               (PlaneType + 1) % 3 );

    // Clean up
    BackList.clear();
    BackAreaList.clear();

    // Success!
    return true;
}
```

This completes the build code for the kD-tree, the last tree type we will have to compile in this lesson. We have now learned how to build quad-trees, oct-trees and kD-trees. These trees should turn out to be very useful throughout your game making career. Indeed, we will be using such trees in virtually all of our lab projects moving forward.

14.20.3 The kD-Tree Query Methods

Querying the kD-tree can be done in a nearly identical manner to quad-trees and oct-trees, with the exception that only two children will need to be tested and traversed during leaf collection. However, although CollectLeavesAABB (and its recursive helper function) will be implemented in an almost

identical manner to the previous versions we have covered for other trees, we have chosen to implement the CollectLeavesRay method (and its recursive helper function) to test using the node split plane instead of the bounding box. This need not be done this way because the box could be used instead, but it does give us an opportunity to introduce a strategy that will be used by its BSP counterpart (BSP trees are generally traversed using the node planes). This will help prepare you for the querying operations we will perform on BSP trees, where the leaves do not fit neatly into axis aligned bounding boxes as we have seen with the trees in this lesson.

CollectLeavesAABB - CKDTree

This method is identical to those implemented in all other tree classes. It simply wraps the call to the recursive CollectAABBRecurse method for the root node. It is called by the application to start the traversal at the root node, finding and collecting all leaves that intersect the query volume.

```
bool CKDTree::CollectLeavesAABB( LeafList & List,
                                 const D3DXVECTOR3 & Min,
                                 const D3DXVECTOR3 & Max )
{
    // Trigger Recursion
    return CollectAABBRecurse( m_pRootNode, List, Min, Max );
}
```

CollectAABBRecurse – CKDTree

This method is called by the previous method to visit the nodes of the tree and step into any children that are inside (or partially inside) the query volume. It is implemented in an identical manner to the other versions of this method for the other tree types with the exception that when a node is not a leaf, we step into only two children.

```
bool CKDTree::CollectAABBRecurse( CKDTreeNode * pNode,
                                  LeafList & List,
                                  const D3DXVECTOR3 & Min,
                                  const D3DXVECTOR3 & Max,
                                  bool bAutoCollect /* = false */ )
{
    bool bResult = false;

    // Validate parameters
    if ( !pNode ) return false;

    // Does the specified box intersect this node?
    if ( !bAutoCollect && !CCollision::AABBIntersectAABB( bAutoCollect,
                                                          Min,
                                                          Max,
                                                          pNode->BoundsMin,
                                                          pNode->BoundsMax ))
           return false;
```

```
    // Is there a leaf here, add it to the list
    if ( pNode->Leaf ) { List.push_back( pNode->Leaf ); return true; }

    // Traverse down to children
    if ( CollectAABBRecurse( pNode->Front, List, Min, Max, bAutoCollect ))
         bResult = true;

    if ( CollectAABBRecurse( pNode->Back , List, Min, Max, bAutoCollect ))
         bResult = true;

    // Return the 'was anything added' result.
    return bResult;
}
```

Once again we use the bAutoCollect boolean to initiate immediate collection of children without further AABB testing.

CollectLeavesRay - CKDTree

The CollectLeavesRay method is also identical to previous versions. It wraps the call to the CollectRayRecurse method for the root node. The function is passed an empty leaf list and a ray origin and delta vector. On function return the passed leaf list will contain a list of all the leaves intersected by the ray.

```
bool CKDTree::CollectLeavesRay( LeafList & List,
                                const D3DXVECTOR3 & RayOrigin,
                                const D3DXVECTOR3 & Velocity )
{
    return CollectRayRecurse( m_pRootNode, List, RayOrigin, Velocity );
}
```

CollectRayRecurse - CKDTree

Unlike prior versions, we will implement this query using the split plane stored at each node instead of the bounding box. These are the types of queries we will be implementing when we start relying on BSP trees in the later lessons of this course. It also tends to be more efficient to perform a single ray plane test rather than a ray box test.

The basic process is to send a ray into a node and perform a ray/plane intersection with the split plane stored there. If the ray is contained completely in the front space of the plane, the function calls itself recursively passing the ray into the front child. If the ray is contained in the back space of the node we send the ray into the back child. However, if the ray is spanning the node we do *not* simply pass it down both the front and back children since this would certainly return leaves that the ray is not intersecting. Instead we will split the ray on the plane and send each ray fragment into its respective child node. Of course, the ray may get split into many fragments during its journey through the tree, but eventually

these ray fragments will pop out in leaf nodes for all leaves that were intersected by the original ray. Figure 14.87 depicts the basic ray splitting concept.

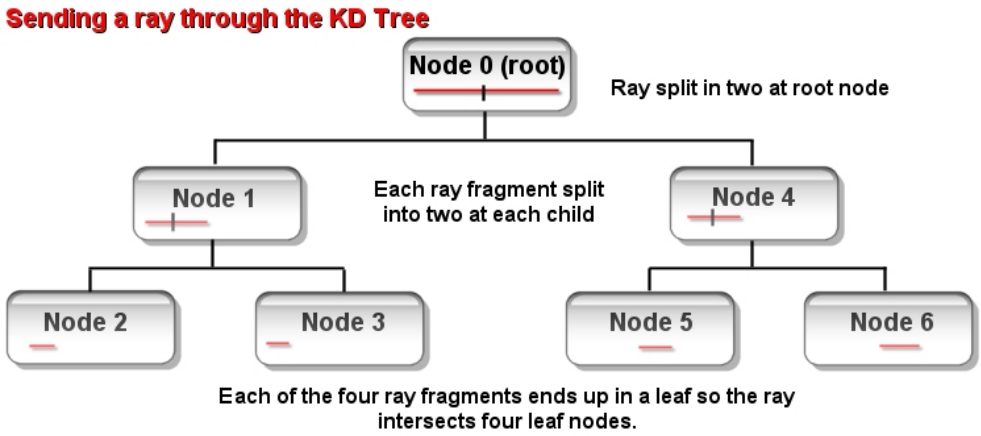


Figure 14.87

In this simple example, the kD-tree consists of seven nodes in total, with four of them being leaf nodes. The ray is assumed to intersect all leaf nodes in this example. Let us step through what is happening.

At the root node, the ray is classified against the plane and is found to be spanning that plane. The ray is split into two segments by this split plane creating two new rays, one in the plane's back space and the other in the front space. The ray fragment in the root node's front space is passed down into the front node (node 1). The ray fragment in node 1 is also found to be spanning its plane, so is further split into two new fragments. One fragment is passed to its front child (node 2) and the other is passed into its back child (node 3). When we reach node 2 and node 3, we find that the ray fragment has been passed into leaf nodes, so the original ray must intersect these leaves. The leaves for both nodes would thus be added to the leaf collection list. At this point the function would return and the recursive process will unroll back up the root node's traversal into its front child. Now the root node has to pass the other ray fragment (the back space fragment) into its back child (node 5). The ray fragment passed into node 5 is also spanning node 5's plane, so it is further split into two ray fragments which end up being passed into leaf nodes 6 and 7 for the front and back fragment, respectively. These are leaves are also added to the leaf collection list.

The above example had the ray was spanning the plane at every node to get across the idea of the recursive splitting process. However, if at any node the ray is found to be in *either* the front or back space, the ray is not split; it is simply passed into the respective child unaltered. In the end, we are essentially just sending the ray down the tree and collecting any leaves that contain any portion/fragment of the ray.

Clipping a ray that spans a plane into two fragments is very easy to do using the CCollision::RayIntersectPlane method we added to our collision library in the previous chapter. This function returns a *t* value of intersection that can be multiplied with the ray delta vector and added to the origin to calculate the actual point of intersection. Once we have the ray start and end points and the ray intersection point, it is easy to see which points define each ray fragment (see Figure 14.88).

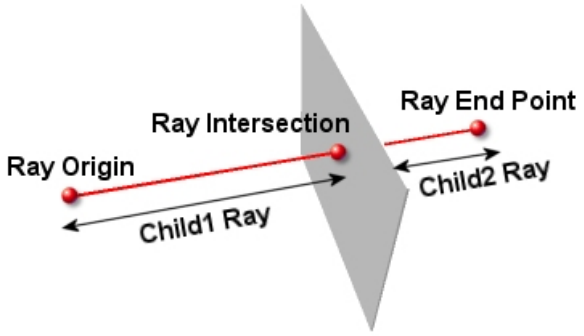


Figure 14.88

In this example we see a plane that intersects a ray. Once the intersection point with the plane is calculated, the ray in the left halfspace in this image is formed by the ray origin and the intersection point, and the ray fragment in the right halfspace is formed by the ray intersection point and the ray end point. That is, the ray intersection point forms the end of one fragment and the beginning of another. These two ray fragments would then be passed into the child nodes.

While splitting the ray and sending the fragments down the tree is pretty simple to implement, what might not be so obvious is why we need to split the ray at all. One might imagine that if the ray spans the plane we could simply send it in its entirety into both children. However, this will actually return incorrect leaves as shown in Figure 14.89.

Figure 14.89 shows a very simple kD-tree consisting of four leaf nodes and a ray that intersects three of those four leaves. In this example, we can assume that the split plane stored at the root node is labeled 'A'. During the building process it was determined that the child node attached to the front of the root (plane A) no longer needed to be subdivided, so the node was made a leaf. This leaf is labeled Leaf 1. When the building process processed the back child of the root (the space down the back of plane A) it was decided that this space should be further subdivided and as such a non-terminal node was created whose split plane is labeled 'B'. Down the front of B it was decided that we no longer needed to partition space, so Leaf 2 was created there. Down the back of B, another non-terminal node was created with the split plane labeled 'C', which further partitioned the top right quadrant of the tree into Leaf 3 and Leaf 4.

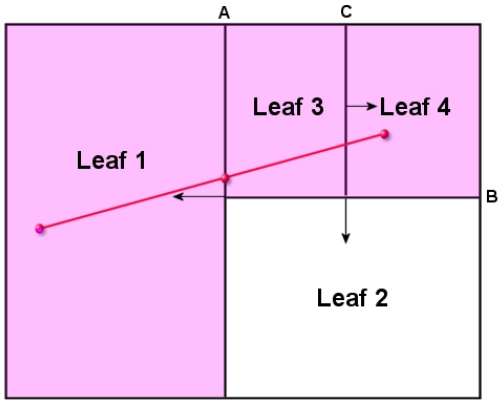


Figure 14.89

We can clearly see that the ray is contained in only three of the four leaf nodes (1, 3, and 4). However, let us see what would happen if we did not split the ray in the spanning case and just sent it through the tree unaltered. What we will learn is that Leaf 2 will be added to the collection list even though the ray does not intersect it. If Leaf 2 contains tens of thousands of polygons, that is a lot of polygons we would need to send to a narrow phase unnecessarily. Let us see why this is the case.

First we send the ray into the root node where it is classified against plane A. We find that it is spanning plane A (remember, we are not splitting in this example to demonstrate the flaw in that approach) so we send the ray into the front and back child of A. Down the front of A the ray ends up in Leaf 1 and it gets added to the list. So far, so good. Down the back of A we find we are at node B and once again classify the ray against plane B. Remembering that planes are infinite, we can see that the ray is actually spanning plane B (imagine B extending infinitely to the left and right). Because it is spanning B, we need to pass it into both children of B. Unfortunately, the front child of B is Leaf 2. Thus, we have just

allowed the ray to fall into Leaf 2 even though it does not intersect this leaf. We need go no further; already we can see that our approach is flawed.

Let us now see what happens if we split the ray at every spanning plane.

Looking again at Figure 14.89 we can see that when the ray is first passed into the root node, it is found to be spanning node plane A. We calculate the intersection point with the plane and create two child rays. The first ray is in the plane's front space and has the original ray's origin as its origin and the intersection point as its end point. The second ray is the ray contained in the plane's back space and has the intersection point as its origin and the original ray end point as its end point. We can see this relationship in Figure 14.90.

Figure 14.90 shows the intersection point between the ray and plane A. We create two new rays at this node. The ray fragment in the plane's front half space is fed into Leaf 1 and Leaf 1 is added to the list. The ray in the plane's back space is created from the ray intersection point with plane A and the original ray end point at the top right of the ray. When we feed this ray fragment into the back space of plane A, we find ourselves at the non-terminal node containing plane B. The important point here is that when we test the ray fed into node B, whose origin is the intersection point between the original ray and plane A, we see that the ray is completely behind plane B and therefore, we never traverse into the front space of B. Therefore, the ray never reaches Leaf 2 and Leaf 2 is never added to the collection list.

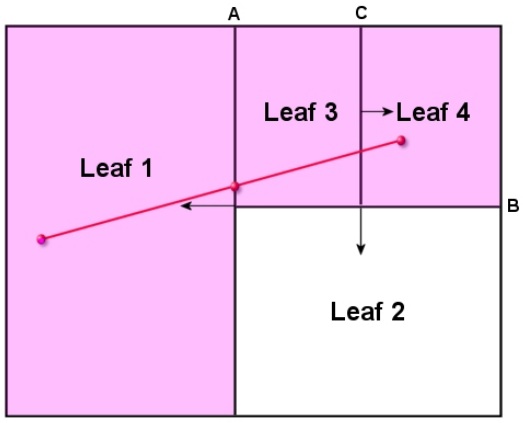


Figure 14.90

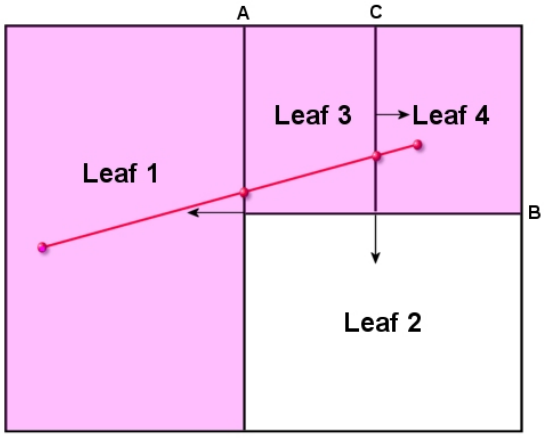


Figure 14.91

In Figure 14.90 the ray fragment was found to be completely behind plane B so it is passed into its child where it enters the non-terminal node containing node C. At node C, the ray spans its plane so the fragment is further clipped into two more ray fragments (see Figure 14.91).

The ray fragment in the back space of C is constructed from the original intersection point between the ray and node A and the intersection of the current fragment with node C. This is passed into the back node of C where it enters Leaf 3 and is added to the leaf collection list. The ray fragment in the front space of C is created from the intersection point of the ray fragment with the split plane of node C and the end point of the ray fragment passed into that function (the original ray end point).

So we have seen why the ray must be split at each node plane. This is a practice that we will see used again when working with BSP trees. Let us now look at the recursive function called by the CKDTree::CollectLeavesRay method to perform this ray splitting traversal through the tree.

Just like the ray traversal method for the other tree types, the parameters are the current node being visited, the leaf list for collection, and the ray origin and delta vectors.

```
bool CKDTree::CollectRayRecurse( CKDTreeNode * pNode,
                                 LeafList & List,
                                 const D3DXVECTOR3 & RayOrigin,
                                 const D3DXVECTOR3 & Velocity )
{
    CCollision::CLASSIFYTYPE PointA, PointB;
    D3DXVECTOR3 RayEnd, Intersection;
    bool        bResult;
    D3DXPLANE   Plane;
    float       t;

    // No operation if the node was null
    if ( !pNode ) return false;

    // If this node stores a leaf, just add it to the list and return
    if ( pNode->Leaf )
    {
        // Add the leaf to the list
        List.push_back( pNode->Leaf );

        // We collected a leaf
        return true;

    } // End if stores a leaf
```

The section of code above shows what happens when the node being visited is a leaf node. This can only happen if a fragment of the original ray has made it into a leaf. When this is the case the node's leaf pointer is simply added to the leaf collection list and the function returns true so that the caller knows that at least part of the ray was found to exist in at least one leaf.

The remainder of the code is executed only when a non-terminal node is entered. The first thing we do is add the ray delta vector to the ray origin to calculate the ray end position. We also use a local variable to point to the node plane for ease of access. We then classify the ray origin and ray end points against the split plane and store the classification results in the PointA and PointB local variables.

```
    // Calculate the end point of the ray
    RayEnd = RayOrigin + Velocity;

    // Retrieve the plane, and classify the ray points against it
    Plane  = pNode->Plane;

    PointA = CCollision::PointClassifyPlane( RayOrigin,
                                             (D3DXVECTOR3&)Plane,
                                             Plane.d );
```

```
    PointB = CCollision::PointClassifyPlane( RayEnd   ,
                                             (D3DXVECTOR3&)Plane,
                                             Plane.d );
```

If both points of the ray lay on the plane then the ray lies exactly on that plane. In this case, we will pass the ray into the front and back children. If a ray is on the plane, it is on the boundary of both nodes, so we should pass it into both nodes. Notice that we initially set the result boolean to false and set it true if any of the front and back recursions return true. We then return this boolean from the function so that the intersection status is returned back up to the root.

```
    // Test for the combination of ray point positions
    if ( PointA == CCollision::CLASSIFY_ONPLANE &&
         PointB == CCollision::CLASSIFY_ONPLANE )
    {
        // Traverse down the front and back
        bResult = false;
        if ( CollectRayRecurse( pNode->Front,
                                List,
                                RayOrigin,
                                Velocity ) ) bResult = true;

        if ( CollectRayRecurse( pNode->Back,
                                List,
                                RayOrigin,
                                Velocity ) ) bResult = true;
        return bResult;

    } // End If both points on plane
```

If the ray origin is in front of the plane and the ray end point is behind the plane (i.e., we have a spanning case), we perform a ray/plane intersection to calculate the *t* value from the ray origin. Once we have the *t* value we calculate the intersection point by scaling the ray delta vector by *t* and adding the result to the ray origin. Once we have the intersection point we recurs into the front child with the sub-ray constructed from the ray origin and the intersection point and recurs into the back child using the sub-ray constructed from the ray intersection point and the ray end point. If either function returns true, we set the boolean result to true and return.

```
    else

    if ( PointA == CCollision::CLASSIFY_INFRONT &&
         PointB == CCollision::CLASSIFY_BEHIND )
    {
        // The ray is spanning the plane, with the origin in front.
        CCollision::RayIntersectPlane( RayOrigin,
                                       Velocity,
                                       (D3DXVECTOR3&)Plane,
                                       Plane.d,
                                       t,
                                       true );

        Intersection = RayOrigin + (Velocity * t);

        // Traverse down both sides passing the relevant segments of the ray
```

```
        bResult = false;

        if ( CollectRayRecurse( pNode->Front,
                                List,
                                RayOrigin,
                                Intersection - RayOrigin ) ) bResult = true;

        if ( CollectRayRecurse( pNode->Back,
                                List,
                                Intersection,
                                RayEnd - Intersection ) ) bResult = true;
        return bResult;

    } // End If Spanning with origin in front
```

The next case is almost identical to the previous spanning case but is executed if the ray origin is behind the plane and the ray end point is in front of the plane. The same basic steps are taken, but the sub-ray fed into each halfspace is constructed differently due the different orientation of the plane with respect to the ray. This time, the sub-ray that we feed into the front node starts at the original ray end point and has the intersection point as its end. The ray we feed into the back child starts at the intersection point and has the original ray origin as its end point.

```
    else

    if ( PointA == CCollision::CLASSIFY_BEHIND &&
         PointB == CCollision::CLASSIFY_INFRONT )
    {
        // The ray is spanning the plane, with the origin in front.
        CCollision::RayIntersectPlane( RayOrigin,
                                       Velocity,
                                       (D3DXVECTOR3&)Plane,
                                       Plane.d,
                                       t,
                                       true );

        Intersection = RayOrigin + (Velocity * t);

        // Traverse down both sides passing the relevant segments of the ray
        bResult = false;

        if ( CollectRayRecurse( pNode->Front,
                                List,
                                Intersection,
                                RayEnd - Intersection ) ) bResult = true;

        if ( CollectRayRecurse( pNode->Back,
                                List,
                                RayOrigin,
                                Intersection - RayOrigin ) ) bResult = true;
        return bResult;

    } // End If Spanning with origin in front
```

The next cases are easier. If both points are in front of the plane, then the ray is totally contained in the plane's front space and we just pass the ray down the front child. Notice that we are actually saying that if Point A or Point B is in front of the plane, we send the entire ray down the front. Remember though, we only get here if the ray is *not* spanning the plane, so this tells us that if Point A or Point B is in the front space then the other must be in the front space too or on the plane (which also counts as the ray being fully contained in the front space).

```
    else

    if ( PointA == CCollision::CLASSIFY_INFRONT ||
         PointB == CCollision::CLASSIFY_INFRONT )
    {

        // Either of the points are in front (but not spanning), pass down front
        return CollectRayRecurse( pNode->Front, List, RayOrigin, Velocity );

    } // End if either point in front
```

Finally, the last code block only gets executed if the ray is behind the plane. That is, if one of the points is definitely behind the plane and the other is either behind the plane or situated on it. When this is the case we pass the entire ray down to the back child.

```
    else
    {
        // Either of the points are behind (but not spanning), pass down  back
        return CollectRayRecurse( pNode->Back, List, RayOrigin, Velocity );

    } // End if either point behind
}
```

Although the coverage of that function might have seemed more laborious than the simple box test, it was good practice to work with a recursive ray splitting procedure like this. We will see such ideas again when working with BSP trees in the coming lessons, so it was worth reviewing. Of course, functions like this can be hard to follow along with in your head (or even on paper) once the tree gets more than a few levels deep, but just remember that all we are doing here is a ray/plane test and creating two rays using the three points at our disposal (the ray origin, the ray intersection point, and the ray end point). We then pass these rays down to the children and eventually all fragments will pop out in the leaves in which they belong. Whenever a fragment enters a leaf, we add the leaf to the list and thus, track and return all leaves that are intersected by the ray.

14.20.4 The kD-Tree Debug Draw Routine

We will not spend time showing the code to the CKDTree::DebugDraw method even though it has been implemented in the source code. If you study the code you will see that it is basically identical to its quad-tree and oct-tree counterparts with the exception that it steps into two children rather than of four or eight.

This concludes our discussion and implementation of spatial trees for this lesson. We now have four tree types (quad-tree, YV quad-tree, oct-tree, and kD-tree) to choose from for use as a spatial manager in future applications. The remainder of this lesson will discuss the addition of a broad phase to our collision system using any of the ISpatialTree derived classes we introduced. As far the collision system (and our application) is concerned, these trees are all the same (they are ISpatialTrees) and thus can be used interchangeably without the user needing to understand the underlying data types.

14.21 Adding Spatial Trees to Lab Project 14.1

Lab Project 14.1 will include the core source code for this lesson and the next. Thus, much of the code in this project associated with the rendering system will be explained in the following lesson. In this lesson, we will focus our attention on how the spatial tree is populated with polygon data and how the collision system uses it to perform very efficient collision queries. Before we discuss the changes to the collision system, we will look at some functionality that has been added to other modules to support the introduction of spatial trees to our existing framework.

14.21.1 The CGameApp Class

Earlier in this lesson we discussed how our collision system will need to make sure that it does not send the same polygon into the expensive narrow phase multiple times. It avoids this by storing a value in each polygon as and when it processes it. As the value it stores is different every time a new query is issued, the collision system knows that if a polygon it is about to test has the same value stored in its structure, it must be one that has been processed already in this frame (perhaps it exists in a leaf that has been previously tested).

Although the system used will be fleshed out in much more detail when we discuss the change to the collision code, we have decided that this functionality might be useful for other modules as well. Therefore, we will store this value in CGameApp and expose two functions that return and increment this counter.

Excerpt from CGameApp.h

```
ULONG                GetAppCounter       () const { return m_nAppCounter; }
void                 IncrementAppCounter ()       { m_nAppCounter++; }
```

The application counter is a simple ULONG that can be retrieved and incremented via the two methods shown above. Below we see that the current value of the application counter is stored in a new CGameApp member variable called m_nAppCounter.

Excerpt from CGameApp.h

```
ULONG                m_nAppCounter;      // Simple Application counter
```

Later you will see why it is useful to have an application global value that can be incremented and retrieved in this way.

14.21.2 The CScene Class

The CScene class has now had an ISpatialTree pointer added to its list of members. It is called m_pSpatialTree and will contain the pointer to the derived tree class we are currently using in our application. Because this is a pointer to a base class, we can make this point at any of our derived tree classes (CQuadTree, COctTree, etc.). Thus, we can easily plug in any of our tree classes without changing any code.

Excerpt from CScene.h

```
ISpatialTree       *m_pSpatialTree;                // Spatial partitioning tree.
```

This pointer is set to NULL in CScene's constructor, as shown below.

Excerpt from CScene::CScene

```
m_pSpatialTree     = NULL;
```

LoadSceneFromIWF – CScene

The LoadSceneFromIWF function is the parent function that loads and processes the data loaded stored in an IWF file. We will not show all the code again here, so you may want to open the source file to check it out for yourself. What we will look at below is a shortened layout of this function just so you can see how the spatial tree is allocated, built, and registered with the collision system.

Excerpt from CScene::LoadSceneFromIWF

```
m_pSpatialTree = new COctTree( m_pD3DDevice, m_bHardwareTnL );

…
…
… Call Process Functions
…
…
…

if ( !m_pSpatialTree->Build( ) ) return false;

m Collision.SetSpatialTree( m pSpatialTree );
…
…
… Create Sound Manager Here
…
…
```

Near the top of the function we allocate a spatial tree of the type we wish to use. In this example, our COctTree class is selected as the spatial manager, but it could have been any one of our derived tree types.

The next section of the function loads the data and calls a series of processing functions to extract the information from the geometry and material vectors populated by the CFileIWF::Load method. One such function that has been changed slightly is the CScene::ProcessVertices method, which is called from CScene::ProcessMeshes (called from LoadSceneFromIWF) for each face loaded from the IWF file. The ProcessVertices method is passed the polygon (as an iwfSurface object) and stores its vertex data in a way that the application can parse and use. Our implementation of this function will create a new CPolygon and populate it with the vertex data for the face currently being loaded. It will then use the ISpatialTree::AddPolygon method to add that polygon to the spatial tree.

After the processing methods have been called, all static polygon data will have been added to the tree, so we issue a call to the tree's Build method, which as we now know, compiles the tree. After the Build function returns, our tree is ready to be used by the collision system, so we call the collision system's new SetSpatialTree method which stores a pointer to this tree for later use in collision queries. We will discuss how the collision system uses the spatial tree in a moment. The rest of the function is unchanged.

ProcessVertices - CScene

We have made some small changes in this function so that the vertex data for the passed face is stored in a new CPolygon structure and added to the spatial tree. Let us have a look at the code.

In the first section of the function we allocate a new CPolygon structure and set its normal to the normal of the passed face. We also set the attribute ID of the polygon to that passed into the function by ProcessMeshes. If the BackFace boolean is set to true, it means we would like to reverse the winding order of this polygon before we use it, so we negate the normal.

```
bool CScene::ProcessVertices( iwfSurface * pFilePoly,
                              ULONG nAttribID,
                              bool BackFace /* = false */ )
{
    // Validate parameters
    if ( !pFilePoly ) return false;

    long      i, nOffset = 0;
    CVertex * pVertices = NULL;
    float     fScale = 1.00f;

    // Allocate a new empty polygon
    CPolygon * pPolygon = new CPolygon;
    if ( !pPolygon ) return false;

    // Set the polygon's attribute ID
    pPolygon->m_nAttribID = nAttribID;
    pPolygon->m_vecNormal = (D3DXVECTOR3&)pFilePoly->Normal;

    if ( BackFace ) pPolygon->m_vecNormal = -pPolygon->m_vecNormal;
```

We then extract the number of vertices in the passed face and use this to inform the CPolygon to allocate enough space in its vertex array for the correct number of vertices.

```
    // Allocate enough vertices
    if ( pPolygon->AddVertex( pFilePoly->VertexCount ) < 0 ) return false;
    pVertices = pPolygon->m_pVertex;
```

Now we loop through each vertex and copy its data into the CPolygon vertex array.

```
    // If we are adding a back-face, setup the offset
    if ( BackFace ) nOffset = pFilePoly->VertexCount - 1;

    // Loop through each vertex and copy required data.
    for ( i = 0; i < (signed)pFilePoly->VertexCount; i++ )
    {
        // Copy over vertex data
        pVertices[i + nOffset].x      = pFilePoly->Vertices[i].x * fScale;
        pVertices[i + nOffset].y      = pFilePoly->Vertices[i].y * fScale;
        pVertices[i + nOffset].z      = pFilePoly->Vertices[i].z * fScale;

        pVertices[i + nOffset].Normal = D3DXVECTOR3&)pFilePoly->Vertices[i].Normal;

        if ( BackFace ) pVertices[i+nOffset].Normal=-pVertices[i + nOffset].Normal;

        // If we have any texture coordinates, set them
        if ( pFilePoly->TexChannelCount > 0 && pFilePoly->TexCoordSize[0] == 2 )
        {
            pVertices[i + nOffset].tu = pFilePoly->Vertices[i].TexCoords[0][0];
            pVertices[i + nOffset].tv = pFilePoly->Vertices[i].TexCoords[0][1];

        } // End if has tex coordinates

        // If we're adding the backface, decrement the offset
        // Remember, we decrement by two here because 'i' will increment
        if ( BackFace ) nOffset -= 2;

    } // Next Vertex
```

Now that we have a CPolygon representation of the face loaded from the file, we add it to the spatial tree.

```
    // Add this new polygon to the spatial tree
    if ( !m_pSpatialTree->AddPolygon( pPolygon ) ) {delete pPolygon;
                                                    return false; }

    // Success!
    return true;
}
```

These are the only changes to the application framework with respect to populating and building the tree. Next we will discuss the changes to the collision system that need to be made so that we finally have a broad phase in our collision detection code.

14.22 Adding a Broad Phase to CCollision

The collision system will not require significant changes in order to include a broad phase. Most of the changes will involve the addition of new functions to collect tree data. You will recall from the previous chapter that the application calls the CCollision::CollideEllipsoid method to perform a collision query. This method transformed the passed ellipsoid into eSpace and then entered a loop that repeatedly called the EllipsoidIntersectScene method to perform collision detection. If the function returned false, then the ellipsoid could move to the requested position, otherwise a response vector was generated and the collision detection step was called again.

Inside the EllipsoidIntersectScene method is where the detection between the swept sphere and the collision geometry was determined. Detection was performed against three pools of geometry using the EllipsoidIntersectBuffers method for the core intersection determination in all three cases. The first pool we tested was any terrain geometry that may have been registered. We did not store the terrain triangles in the collision system's static geometry array, but instead generated them on the fly using a temporary buffer that was passed to the EllipsoidIntersectBuffers method. After recording any intersections with terrain triangles, we then iterated through the collision system's dynamic object array passing the geometry buffers of each dynamic object into the EllipsoidIntersectBuffers method to record and store any collisions that occurred. Finally, we passed the collision system's static geometry buffers into the EllipsoidIntersectBuffers method so that the static polygon data used by the system was also tested and any collisions recorded.

The EllipsoidIntersectScene method will now be updated in two places. First, there will be a fourth pool of geometry that needs to be checked -- the geometry in the spatial tree that has been assigned to the collision system. As we have seen, the application populates and builds this tree using static scene data and then assigns it to the tree using the CCollision::SetSpatialTree method (which just caches the pointer in a member variable). During the EllipsoidIntersectScene method, we will test the geometry in the tree by sending the AABB of the swept sphere into the spatial tree's CollectLeavesAABB method. Once all intersecting leaves are returned, we will test each polygon in those leaves to see if their bounding boxes intersect the bounding box of the swept sphere (an additional broad phase step). We will only perform narrow phase intersection for polygons that have a good potential for intersect the sphere's AABB. This is essentially our broad phase implementation for static geometry.

One thing might be bothering you at this point. If the static polygon data that we load will now be stored in the tree, what happens to the original static geometry database in the collision system? Well, those vectors will no longer be used because we will always choose to use a spatial tree and connect that to the collision system. We could still add geometry to the collision system's static vectors (using the CCollision::AddIndexedPrimitive method for example) but as we know, every polygon in these vectors will be tested in the narrow phase. Thus, storing static geometry there instead of in our spatial tree would force us to give up our extremely efficient broad phase step. So does that mean that all of the functions that we wrote to store polygons and collide against the static geometry were a waste of time? Absolutely not. Our aim was to create a collision library that can be used in your applications that might rely on other technologies. Although we will favor using a spatial tree to store our static geometry, there may well be users that wish to use the collision system without those trees. Therefore, while it is recommended that you always use a spatial tree to store your static collision data, the collision system

will still do its job even if one is not provided. As we saw in the previous lesson, if somebody just wants to quickly add polygon data to the collision system, we have provided a means to do so.

The other area where the EllipsoidIntersectScene method will change slightly will when testing the swept sphere against dynamic objects. In the previous chapters we saw that we did this by passing the geometry buffers of the dynamic object currently being processed (and its matrix) into the EllipsoidIntersectBuffers method. This essentially performed the narrow phase detection step on every polygon in the dynamic object's buffer. Now, we first test the bounding box of the dynamic object against the AABB of the swept sphere and only call the EllipsoidIntersectBuffers method if the two intersect. Although this may seem like a very crude broad phase for our dynamic objects, AABB/AABB intersection tests are so cheap and our project uses so few dynamic objects there is really no point in using a tree traversal to determine which objects intersect. Therefore, unlike the rendering system that we will cover in the following lesson, our collision system does not use the spatial tree to implement a broad phase for dynamic objects, instead it just loops through each performing an AABB/AABB cull.

> **Note:** It would be easy enough to add some code to take advantage of the tree for your dynamic objects. After all, they are already stored in the tree and you would already have incurred the cost of traversal when you test for static geometry. Thus, there would be only a few extra lines of logic needed to extract the dynamic objects from the leaves you collided with.

In order to implement the AABB/AABB broad phase for the dynamic object testing we will need to extend our collision system's DynamicObject structure to make room for two vectors that will store the minimum and maximum extents of its bounding box. Remembering from the previous lesson that dynamic objects are stored in the collision system as model space geometry with an associated world transformation matrix, the bounding box stored in each dynamic object will also be defined in the model space of the object too. We will see later that when performing the AABB/AABB test between the swept sphere's AABB and the dynamic object, the dynamic object's AABB will be transformed into world space for the test.

Our updated DynamicObject structure is shown below with the two new members highlighted in bold.

Excerpt from CCollision.h

```
struct DynamicObject
{
        CollTriVector  *pCollTriangles;
        CollVertVector *pCollVertices;
        D3DXMATRIX     *pCurrentMatrix;
        D3DXMATRIX      LastMatrix;
        D3DXMATRIX      VelocityMatrix;
        D3DXMATRIX      CollisionMatrix;
        long            ObjectSetIndex;
        bool            IsReference;
        D3DXVECTOR3     BoundsMin;      // Minimum bounding box extents
        D3DXVECTOR3     BoundsMax;      // Maximum bounding box extents
};
```

We just learned that the EllipsoidIntersectScene method will now perform a collision query on the tree. That is, it will use the spatial tree's CollectLeavesAABB method to return a list of all the leaves intersected by the AABB of the swept sphere. Two new members have been added to the CCollision

class for this purpose. The first is a pointer to an ISpatialTree derived object which the application will set using the CCollision::SetSpatialTree. The second is a member of type ISpatialTree::LeafList. Remember, when we run an AABB query on the tree, the CollectLeavesAABB method expects to be passed an empty leaf list that it will fill with leaf pointers. Therefore, the collision object maintains a leaf list of its own which it can empty each time and pass into this function to collect the leaves whenever a query is performed. Thus, once our collision system has called the ISpatialTree::CollectLeavesAABB method, its m_treeLeafList member will contain a list of all the leaves intersected by the swept sphere's AABB. The new members of CCollision are shown below.

New Members to CCollision

```
ISpatialTree           *m_pSpatialTree;
ISpatialTree::LeafList  m_TreeLeafList;
D3DXVECTOR3             m_vecEllipsoidMin;
D3DXVECTOR3             m_vecEllipsoidMax;
```

Notice that we have also added two new 3D vectors as members in the collision object. Because we will need to access the world space bounding box of the ellipsoid during our EllipsoidIntersectScene method, we use these new members to cache the extents. For example, we will need the world space AABB of the ellipsoid in the methods that send the AABB of the ellipsoid down the tree to collect all intersecting leaves. We will also need the world space AABB during the dynamic object tests when we implement a broad phase AABB/AABB test between the world space AABB of the dynamic object and the world space AABB of the ellipsoid. Rather than continually having to calculate this box each time we need it, we will add a function to calculate the ellipsoid bounds. It will generate a bounding box around the ellipsoid and store it in these two variables for the life of the query. This will allow any helper functions that need this information to be able to quickly access it.

SetSpatialTree - CCollision

One of the new methods that we saw being used earlier assigns the collision object's m_pSpatialTree pointer the tree the application would like it to use. This simple function is called once the application has created and populated the tree with the required data.

```
void CCollision::SetSpatialTree( ISpatialTree * pTree )
{
    // Store the tree
    m_pSpatialTree = pTree;
}
```

CalculateEllipsoidBounds – CCollision

This method calculates the bounding box of the query ellipsoid. It tests the x, y, and z components of two points, the ellipsoid center position and the ellipsoid center position plus the velocity vector, to find the minimum and maximum extents for a bounding box that encompasses both points. We will see this method being used in a few of the other collision functions later on.

The function is passed the ellipsoid center position, its radius vector, and its velocity vector. It first examines the radius vector to determine which of its components is the largest. We will use this extent to essentially represent a sphere that bounds the ellipsoid. Notice at the top of the function how we reference the member variables m_vecEllipsoidMin and m_vecEllipsoidMax (which will receive the resulting AABB) with some local variables for ease of access.

```
void CCollision::CalculateEllipsoidBounds( const D3DXVECTOR3& Center,
                                           const D3DXVECTOR3& Radius,
                                           const D3DXVECTOR3& Velocity )
{
    float       fLargestExtent;

    D3DXVECTOR3& vecMin = m_vecEllipsoidMin;
    D3DXVECTOR3& vecMax = m_vecEllipsoidMax;

    // Find largest extent of ellipsoid
    fLargestExtent = Radius.x;
    if ( Radius.y > fLargestExtent ) fLargestExtent = Radius.y;
    if ( Radius.z > fLargestExtent ) fLargestExtent = Radius.z;
```

At this point we have stored the largest component of the ellipsoid's radius vector in fLargestExtent. We will now initialize the bounding box to impossibly small values before performing the tests.

```
    // Reset the bounding box values
    vecMin = D3DXVECTOR3( FLT_MAX, FLT_MAX, FLT_MAX );
    vecMax = D3DXVECTOR3( -FLT_MAX, -FLT_MAX, -FLT_MAX );
```

The first thing we will do is add the largest extent to the ellipsoid center position and test the result to see if it is greater than the value currently stored in the bounding box maximum extent. We do this on a per-component basis. We are essentially growing the box if the sphere that surrounds the ellipsoid pierces the previously discovered maximum extent along any axis. If it does, that component of the maximum extent is updated with the new maximum.

```
    // Calculate the bounding box extents of where the ellipsoid currently
    // is, and the position it will be moving to.

    if ( Center.x + fLargestExtent > vecMax.x )
      vecMax.x = Center.x + fLargestExtent;

    if ( Center.y + fLargestExtent > vecMax.y )
      vecMax.y = Center.y + fLargestExtent;

    if ( Center.z + fLargestExtent > vecMax.z )
      vecMax.z = Center.z + fLargestExtent;
```

We now do exactly the same test for each component to test that the sphere bounding the ellipsoid does not pierce any previously found minimum extent for the box along any world axis. Once again, if it does, that component of the box is updated to the new minimum.

```
    if ( Center.x - fLargestExtent < vecMin.x )
```

```
      vecMin.x = Center.x - fLargestExtent;

    if ( Center.y - fLargestExtent < vecMin.y )
      vecMin.y = Center.y - fLargestExtent;

    if ( Center.z - fLargestExtent < vecMin.z )
      vecMin.z = Center.z - fLargestExtent;
```

Now repeat the same two tests again, only this time we add the velocity vector and the largest radius extent to the center position of the ellipsoid. What we are essentially doing is building a sphere that bounds the ellipsoid at the very end of its velocity vector.

```
    if ( Center.x + Velocity.x + fLargestExtent > vecMax.x )
      vecMax.x = Center.x + Velocity.x + fLargestExtent;

    if ( Center.y + Velocity.y + fLargestExtent > vecMax.y )
      vecMax.y = Center.y + Velocity.y + fLargestExtent;

    if ( Center.z + Velocity.z + fLargestExtent > vecMax.z )
      vecMax.z = Center.z + Velocity.z + fLargestExtent;

    if ( Center.x + Velocity.x - fLargestExtent < vecMin.x )
      vecMin.x = Center.x + Velocity.x - fLargestExtent;

    if ( Center.y + Velocity.y - fLargestExtent < vecMin.y )
      vecMin.y = Center.y + Velocity.y - fLargestExtent;

    if ( Center.z + Velocity.z - fLargestExtent < vecMin.z )
      vecMin.z = Center.z + Velocity.z - fLargestExtent;
```

Finally, just to add a bit of padding to manage limited floating point precision, we extend the box by one unit along each axis in both directions.

```
    // Add Tolerance values
    vecMin -= D3DXVECTOR3( 1.0f, 1.0f, 1.0f );
    vecMax += D3DXVECTOR3( 1.0f, 1.0f, 1.0f );
}
```

We will see this method being used later when discussing the new additions to the collision system.

14.22.1 Dynamic Object Bounding Box Generation

When a dynamic object is registered with the collision system, it will now have to calculate its model space bounding box so that it can be stored in the DynamicObject structure and used during the dynamic object's broad phase test. Adding such a feature involves simply extending the code of the CCollision::AddBufferData method.

You should recall from the previous chapter how the AddBufferData method was a general purpose utility function that was used to add a passed array of vertices and indices to the passed geometry

vectors. This was useful because the same function could be used to add the vertex data for a static polygon to the collision system's static geometry vectors, or it could be used to add the vertex and index data for a dynamic object to the dynamic object's geometry buffers. The AddBufferData method could even be passed an optional matrix pointer parameter that would be used to transform the vertex data prior to storing it in the passed geometry buffers.

To jog your memory, below we see an example of how this method was used by the CCollision::AddIndexedPrimitive method. This method was called in our previous application for every static polygon loaded from the IWF file. As you can see, this method just issues a call to the AddBufferData method, passing as the first two parameters the static geometry buffers of the collision system. When the function returns, the passed vertices and indices will have been added to the collision system's static vectors. We are also reminded that the current transformation matrix for the collision system is also passed in, so that if the application has set this to anything other than an identity matrix (its default state) the geometry will be transformed before it is added to the collision system's static geometry vectors.

```
bool CCollision::AddIndexedPrimitive( LPVOID Vertices,
                                      LPVOID Indices,
                                      ULONG VertexCount,
                                      ULONG TriCount,
                                      ULONG VertexStride,
                                      ULONG IndexStride,
                                      USHORT MaterialIndex )
{
    ULONG i, BaseTriCount;

    // Store the previous triangle count
    BaseTriCount = m_CollTriangles.size();

    // Add to the standard buffer
    if ( !AddBufferData( m_CollVertices,
                         m_CollTriangles,
                         Vertices,
                         Indices,
                         VertexCount,
                         TriCount,
                         VertexStride,
                         IndexStride,
                         m_mtxWorldTransform ) ) return false;

    // Loop through and assign the specified material ID to all triangles
    for ( i = BaseTriCount; i < m_CollTriangles.size(); ++i )
    {
        // Assign to triangle
        m_CollTriangles[i].SurfaceMaterial = MaterialIndex;

    } // Next Triangle

    // Success
    return true;
}
```

The AddBufferData method was also used to add dynamic objects to the system. In this case, the vertex and index data of the object would be passed as the first two parameters, along with the geometry buffers for the dynamic object. The vertex and index data would be copied into the geometry buffers of the collision system's representation of that dynamic object as a result.

Because our dynamic objects will now need to have bounding boxes calculated for them, the AddBufferData member has had two optional parameters added which are only used when the function is being called for dynamic objects. These two parameters are pointers to the dynamic object's bounding box extents. The function will calculate the bounding box extents and store them in the dynamic object before returning.

Below we see the AddDynamicOject method of CCollision which the application can use to register a single dynamic object with the collision system. This code is virtually unchanged from the previous version with the exception that it now passes the addresses of the dynamic object's bounding box extents as the last two parameters to AddBufferData. All of the other code was discussed in detail over the last two lessons, and will not be discussed here.

```
long CCollision::AddDynamicObject( LPVOID Vertices,
                                   LPVOID Indices,
                                   ULONG VertexCount,
                                   ULONG TriCount,
                                   ULONG VertexStride,
                                   ULONG IndexStride,
                                   D3DXMATRIX * pMatrix,
                                   bool bNewObjectSet /* = true */ )
{
    D3DXMATRIX      mtxIdentity;
    DynamicObject * pObject = NULL;

    // Reset identity matrix
    D3DXMatrixIdentity( &mtxIdentity );

    // Ensure that they aren't doing something naughty
    if ( m_nLastObjectSet < 0 ) bNewObjectSet = true;

    // We have used another object set index.
    if ( bNewObjectSet ) m_nLastObjectSet++;

    try
    {
        // Allocate a new dynamic object instance
        pObject = new DynamicObject;
        if ( !pObject ) throw 0;

        // Clear the structure
        ZeroMemory( pObject, sizeof(DynamicObject) );

        // Allocate an empty vector for the buffer data
        pObject->pCollVertices  = new CollVertVector;
        if ( !pObject->pCollVertices ) throw 0;
        pObject->pCollTriangles = new CollTriVector;
        if ( !pObject->pCollTriangles ) throw 0;
```

```
        // Store the matrices we'll need for
        pObject->pCurrentMatrix  = pMatrix;
        pObject->LastMatrix      = *pMatrix;
        pObject->CollisionMatrix = *pMatrix;
        pObject->ObjectSetIndex  = m_nLastObjectSet;
        pObject->IsReference     = false;
        D3DXMatrixIdentity( &pObject->VelocityMatrix );

        // Add to the dynamic object's database
        if ( !AddBufferData( *pObject->pCollVertices,
                            *pObject->pCollTriangles,
                             Vertices,
                             Indices,
                             VertexCount,
                             TriCount,
                             VertexStride,
                              IndexStride,
                              mtxIdentity,
                              &pObject->BoundsMin,
                              &pObject->BoundsMax ) ) throw 0;

        // Store the dynamic object
        m_DynamicObjects.push_back( pObject );

    } // End try block

    catch (...)
    {
        // Release the object if it got created
        if ( pObject )
        {
            if ( pObject->pCollVertices  ) delete pObject->pCollVertices;
            if ( pObject->pCollTriangles ) delete pObject->pCollTriangles;
            delete pObject;

        } // End if object exists

        // Return failure
        return -1;

    } // End Catch Block

    // Return the object index used
    return m_nLastObjectSet;
}
```

When AddBufferData returns, it will have calculated the extents of the model space AABB and will have stored them in the final two parameters.

You will see this slight modification to the AddBufferData call in all of the methods we discussed which register dynamic objects. You will recall for example that when we register an actor with the collision system (using the CCollision::AddActor method) and pass the parameter that states that we would like the actor to be dynamic, the hierarchy is traversed and a dynamic object is created in the collision

system for each mesh container found. However, each dynamic object created from a mesh container is created in exactly the same way as shown above and will now have their bounding boxes computed.

The new version of AddBufferData is shown below with the added code that calculates and stores the model space AABB of the passed geometry.

The first section of the function is the same, it simply makes room in the passed vertex and triangle buffers to add the new data if the buffers are not currently large enough.

```
bool CCollision::AddBufferData( CollVertVector& VertBuffer,
                                CollTriVector& TriBuffer,
                                LPVOID Vertices,
                                LPVOID Indices,
                                ULONG VertexCount,
                                ULONG TriCount,
                                ULONG VertexStride,
                                ULONG IndexStride,
                                const D3DXMATRIX& mtxWorld,
                                D3DXVECTOR3 * pBoundsMin /* = NULL */,
                                D3DXVECTOR3 * pBoundsMax /* = NULL */ )
{
    ULONG        i, Index1, Index2, Index3, BaseVertex;
    UCHAR       *pVertices = (UCHAR*)Vertices;
    UCHAR       *pIndices  = (UCHAR*)Indices;
    CollTriangle Triangle;
    D3DXVECTOR3  Edge1, Edge2;

    // Validate parameters
    if ( !Vertices || !Indices || !VertexCount || !TriCount ||
         !VertexStride || (IndexStride != 2 && IndexStride != 4) ) return false;

    // Catch template exceptions
    try
    {
        // Grow the vertex buffer if required
        while ( VertBuffer.capacity() < VertBuffer.size() + VertexCount )
        {
            // Reserve extra space
            VertBuffer.reserve( VertBuffer.capacity() + m_nVertGrowCount );

        } // Keep growing until we have enough

        // Grow the triangle buffer if required
        while ( TriBuffer.capacity() < TriBuffer.size() + TriCount )
        {
            // Reserve extra space
            TriBuffer.reserve( TriBuffer.capacity() + m_nTriGrowCount );

        } // Keep growing until we have enough
```

Next we store the current number of vertices in the passed vertex vector so that we know where the indices will start indexing (it might not be empty when the function is called, which is certainly the case

when the function is called to add triangles to the static geometry vectors). We also initialize the passed extent vectors to impossible values.

```
        // Store the original vertex count before we copy
        BaseVertex = VertBuffer.size();

        // Reset bounding box values if provided
        if ( pBoundsMin ) *pBoundsMin = D3DXVECTOR3( FLT_MAX, FLT_MAX, FLT_MAX );
        if ( pBoundsMax ) *pBoundsMax = D3DXVECTOR3( -FLT_MAX, -FLT_MAX, -FLT_MAX);
```

Now we loop through each vertex in the passed array and transform it by the passed transformation matrix. We then test the current bounding box extents against the vertex and grow the box if the vertex is found not to be contained in it. We then add the vertex to the passed vertex vector.

```
        // For each vertex passed
        for ( i = 0; i < VertexCount; ++i, pVertices += VertexStride )
        {
            // Transform the vertex
            D3DXVECTOR3 Vertex = *(D3DXVECTOR3*)pVertices;
            D3DXVec3TransformCoord( &Vertex, &Vertex, &mtxWorld );

            // Calculate bounding box extents
            if ( pBoundsMin )
            {
                if ( Vertex.x < pBoundsMin->x ) pBoundsMin->x = Vertex.x;
                if ( Vertex.y < pBoundsMin->y ) pBoundsMin->y = Vertex.y;
                if ( Vertex.z < pBoundsMin->z ) pBoundsMin->z = Vertex.z;

            } // End if minimum extents variable passed

            if ( pBoundsMax )
            {
                if ( Vertex.x > pBoundsMax->x ) pBoundsMax->x = Vertex.x;
                if ( Vertex.y > pBoundsMax->y ) pBoundsMax->y = Vertex.y;
                if ( Vertex.z > pBoundsMax->z ) pBoundsMax->z = Vertex.z;

            } // End if maximum extents variable passed

            // Copy over the vertices
            VertBuffer.push_back( Vertex );

        } // Next Vertex
```

The last section of the function calculates the indices of each triangle and is unchanged.

```
        // Build the triangle data
        for ( i = 0; i < TriCount; ++i )
        {
            // Retrieve the three indices
            Index1 = ( IndexStride == 2 ? (ULONG)*((USHORT*)pIndices) :
                                          *((ULONG*)pIndices) );
            pIndices += IndexStride;

            Index2 = ( IndexStride == 2 ? (ULONG)*((USHORT*)pIndices) :
```

```
                                        *((ULONG*)pIndices) );
            pIndices += IndexStride;

            Index3 = ( IndexStride == 2 ? (ULONG)*((USHORT*)pIndices) :
                       *((ULONG*)pIndices) );
            pIndices += IndexStride;

            // Store the details
            Triangle.SurfaceMaterial = 0;
            Triangle.Indices[0]      = Index1 + BaseVertex;
            Triangle.Indices[1]      = Index2 + BaseVertex;
            Triangle.Indices[2]      = Index3 + BaseVertex;

            // Retrieve the vertices themselves
            D3DXVECTOR3 &v1 = VertBuffer[ Triangle.Indices[0] ];
            D3DXVECTOR3 &v2 = VertBuffer[ Triangle.Indices[1] ];
            D3DXVECTOR3 &v3 = VertBuffer[ Triangle.Indices[2] ];

            // Calculate and store edge values
            D3DXVec3Normalize( &Edge1, &(v2 - v1) );
            D3DXVec3Normalize( &Edge2, &(v3 - v1) );

            // Skip if this is a degenerate triangle, we don't want it in our set
            float fDot = D3DXVec3Dot( &Edge1, &Edge2 );
            if ( fabsf(fDot) >= (1.0f - 1e-5f) ) continue;

            // Generate the triangle normal
            D3DXVec3Cross( &Triangle.Normal, &Edge1, &Edge2 );
            D3DXVec3Normalize( &Triangle.Normal, &Triangle.Normal );

            // Store the triangle in our database
            TriBuffer.push_back( Triangle );

        } // Next Triangle

    } // End Try Block

    catch ( ... )
    {
        // Just fail on any exception.
        return false;

    } // End Catch Block


    // Success!
    return true;
}
```

That covers all of the changes involved in registering geometry with the collision system. They are very small, but the calculation of the bounding boxes of dynamic objects is a significant step that should not be overlooked. We can now move on to discussing the changes to the core collision system.

14.22.2 Changes to the Collision Query Engine

When the application performs a query, it calls the CCollision::CollideEllipsoid method. This method is responsible for transforming the ellipsoid into eSpace and then setting up a loop that will call the detection method (EllipsoidIntersectScene) and generate a slide vector every time the detection function returns true for intersection. Only when the detection phase returns false have we found a new position for the ellipsoid that is free from obstruction.

As we have briefly touched on, we need to make sure during a given detection step that we do not send a polygon to the expensive narrow phase more than once. If we are using a clipped tree, this will never happen since each polygon will be contained in one and only one leaf. However, if a non-clipped tree is being used (often the case) we may have a situation where a single polygon is contained in multiple leaves. Further, when the collision system calls the ISpatialTree::CollectLeavesAABB method, it may get back a list of intersecting leaves that each contains this same polygon. We certainly never want to test the same polygon more than once, so we saw earlier how we implemented an application timer to allow us to efficiently invalidate polygons in the collision system.

The CPolygon structure stores a member called m_AppCounter that can be used to store the current value held in the application counter. Why is this useful?

When our collision system performs a query on the tree, it will get back a list of leaves. Then it will iterate through the polygons in each of those leaves and potentially pass them on to the narrow phase if we have an AABB/AABB intersection between the polygon's box and the swept sphere's box. Once a polygon has been tested and is about to be passed to the narrow phase, we will store the current value of the application counter in its m_nAppCounter member. We will only send polygons to the narrow phase whose m_nAppCounter member is not equal to the current value of the application counter. If the application counter matches, it means this polygon has already been processed since the application counter was last incremented. This would be the case if we were processing a polygon in a leaf that had existed in a previous leaf we had tested. By the end of the collision step, all the polygons in all the returned leaves will have had their m_nAppCounter members set to the same value as the current application counter.

While we could achieve the same end by storing a boolean in each polygon which is set to true once it has been processed, this would leave us with the very unpleasant (and slow) task of looping through every polygon and resetting their booleans prior to performing a new query. Since our CollideEllipsoid method may perform many detection queries during a single update, it would become prohibitively expensive if tens of thousands of polygons had to have their booleans reset dozens of times. Fortunately, with our counting logic, all we have to do before performing another detection query is increment the application's main counter. At this point, the application counter will no longer match the m_nAppCounter member in any polygon, so they will all be tested in the next update.

Of you open up the source code to the CCollision::CollideEllipsoid method you will see that we have added a new function call that increments the application counter with each iteration of the detection loop. We will not show the entire function here since we have only added one new line. However, the

following listing demonstrates where this new line (actually two new lines which are both the same) exists. The new lines are highlighted in bold.

Excerpt from CCollision::CollideEllipsoid

```
bool CCollision::CollideEllipsoid( const D3DXVECTOR3& Center,
                                   const D3DXVECTOR3& Radius,
                                   const D3DXVECTOR3& Velocity,
                                   D3DXVECTOR3& NewCenter,
                                   D3DXVECTOR3& NewIntegrationVelocity,
                                   D3DXVECTOR3& CollExtentsMin,
                                   D3DXVECTOR3& CollExtentsMax )
{
      …
      … Calculate ESpace ellipsoid start, end and radius vectors here
      …
      …
    // Keep testing until we hit our max iteration limit
    for ( i = 0; i < m_nMaxIterations; ++i )
    {

        if ( EllipsoidIntersectScene( eFrom,
                                      Radius,
                                      eVelocity,
                                      m_pIntersections,
                                      IntersectionCount,
                                      true,
                                      true ) )
        {
            …
            … Calculate Response vector and eFrom here
            …
            bHit=true;

        } // End if we got some intersections
        else
        {
            // We found no collisions, so break out of the loop
            break;

        } // End if no collision

        // Increment the app counter so that our polygon testing is reset
        GetGameApp()->IncrementAppCounter();

    } // Next Iteration

    // Did we register any intersection at all?
    if ( bHit )
    {
        // Increment the app counter so that our polygon testing is reset
        GetGameApp()->IncrementAppCounter();

        …
        …
        …  If we ran out of iterations to one collision test here and just
        …  return first intersecting position
```

```
        …


    } // End if intersection found

    // Store the resulting output values
    NewCenter              = vecOutputPos;
    NewIntegrationVelocity = vecOutputVelocity;

    // Return hit code
    return bHit;
}
```

As you can see, the application counter is advanced after each collision detection query so that the next time it is performed, the polygon counters no longer match the application counter and the polygons are considered again in the next test.

EllipsoidIntersectScene - CCollision

The EllipsoidIntersectScene method is called by CollideEllipsoid each time a detection test has to be performed on a velocity vector (or slide vector). Previously this method tested three pools of data; the terrain data, the dynamic objects, and the static geometry vectors. We will now add an additional section that performs a collision test on the static geometry stored in the spatial tree (if one has been assigned to the collision system). In our application, all static data will be stored in the spatial tree and the static geometry vectors of the collision system will be left empty. We also add the AABB/AABB broad phase to the dynamic object test. Although most of the code is unchanged, we will still show the entire function again, but only spend time discussing the additional code in any detail.

The first section of the function transforms the passed ellipsoid centerpoint and velocity vector into eSpace if the caller did not pass these in as eSpace parameters to begin with. It also sets the first collision intersect time (*t*) to 1.0 (i.e., the end of the vector and thus no collision detected).

```
bool CCollision::EllipsoidIntersectScene( const D3DXVECTOR3 &Center,
                                          const D3DXVECTOR3& Radius,
                                          const D3DXVECTOR3& Velocity,
                                          CollIntersect Intersections[],
                                          ULONG & IntersectionCount,
                                          bool bInputEllipsoidSpace /* = false */,
                                          bool bReturnEllipsoidSpace /* = false */)
{
    D3DXVECTOR3 eCenter, eVelocity, eAdjust, vecEndPoint, InvRadius;
    float       eInterval;
    ULONG       i;

    // Vectors for terrain building
    CollVertVector VertBuffer;
    CollTriVector  TriBuffer;

    // Calculate the reciprocal radius to prevent the divides we would need
    InvRadius = D3DXVECTOR3( 1.0f / Radius.x, 1.0f / Radius.y, 1.0f / Radius.z );
```

```
    // Convert the values specified into ellipsoid space if required
    if ( !bInputEllipsoidSpace )
    {
        eCenter   = Vec3VecScale( Center, InvRadius );
        eVelocity = Vec3VecScale( Velocity, InvRadius );

    } // End if the input values were not in ellipsoid space
    else
    {
        eCenter   = Center;
        eVelocity = Velocity;

    } // End if the input values are already in ellipsoid space

    // Reset ellipsoid space interval to maximum
    eInterval = 1.0f;

    // Reset initial intersection count to 0 to save the caller having to do this.
    IntersectionCount = 0;

    // Calculate the bounding box of the ellipsoid
    D3DXVECTOR3 vecCenter   = Vec3VecScale( eCenter, Radius );
    D3DXVECTOR3 vecVelocity = Vec3VecScale( eVelocity, Radius );
    CalculateEllipsoidBounds( vecCenter, Radius, vecVelocity );
```

The new additions to this section are highlighted in bold at the bottom of the code. We transform the ellipsoid center position and velocity vector into world space (by scaling it by the radius vector) and then pass the world space position, velocity, and radius vector of the ellipsoid into the CalculateEllipsoidBounds method (discussed earlier). When this method returns, the CCollision::m_vecEllipsoidMin and CCollision::m_vecEllipsoidMax member variables will store the extents of the current world space bounding box of the ellipsoid. We will see why we need this later in the function.

The next section, which is unchanged from the previous lab project, collects triangles from any terrain object in the immediate vicinity of the swept sphere and runs the intersection test.

```
    // Iterate through our terrain database
    TerrainVector::iterator TerrainIterator = m_TerrainObjects.begin();
    for ( ; TerrainIterator != m_TerrainObjects.end(); ++TerrainIterator )
    {
        const CTerrain * pTerrain = *TerrainIterator;

        // Collect the terrain triangle data
        if ( !CollectTerrainData( *pTerrain,
                                  vecCenter,
                                  Radius,
                                  vecVelocity,
                                  VertBuffer,
                                  TriBuffer ) ) continue;

        // Perform the ellipsoid intersect test against this set of terrain data
        EllipsoidIntersectBuffers( VertBuffer,
                                   TriBuffer,
```

```
                                    eCenter,
                                    Radius,
                                    InvRadius,
                                    eVelocity,
                                    eInterval,
                                    Intersections,
                                    IntersectionCount );

        // Clear buffers for next terrain
        VertBuffer.clear();
        TriBuffer.clear();

    } // Next Terrain
```

The next section is new. It contains the code that performs the intersection test against the geometry stored in the spatial tree (if one has been registered with the collision system). The first thing we do is pass the world space ellipsoid center position and velocity vector to a new method called CCollision::CollectTreeLeaves. This method will simply pass the world space bounding box of the ellipsoid into the ISpatialTree::CollectLeavesAABB method. Earlier we learned that the collision system now includes a member of type LeafList, which is also passed to the CollectLeavesAABB method and used to collect the pointers to the leaves that intersect the ellipsoid AABB. We will look at this method in a moment.

When the CollectTreeLeaves method returns, the CCollision::m_TreeLeafList member will currently contain all the pointers to all leaves that intersect the swept sphere's bounding box. We then call another new method called EllipsoidIntersectTree which performs the narrow phase test on the polygon data stored in those leaves. The EllipsoidIntersectTree method is almost identical to the EllipsoidIntersectBuffers method which performs narrow phase detection on other geometry pools. The reason we had to make a special version of this function is that the CPolygon data stored in the tree uses a different format from what the collision system is used to working with. Rather than incur the costs of copying the data over and converting it into the old format on the fly (which can get very expensive!), we just added a modified version of the prior code to work with CPolygons. We only want to test polygons that have bounding boxes that intersect the ellipsoid's bounding box and that have not already been tested in this update. This is another method we will look at in a moment.

```
    // Collide against tree if available
    if ( m_pSpatialTree )
    {

        // Collect the leaf list if necessary
        CollectTreeLeaves( vecCenter, Radius, vecVelocity );

        EllipsoidIntersectTree( eCenter,
                                Radius,
                                InvRadius,
                                eVelocity,
                                eInterval,
                                Intersections,
                                IntersectionCount );

    } // Next Tree
```

When the EllipsoidIntersectTree method returns, any intersections found between the swept sphere and the polygons stored in the tree will be added to the passed Intersections array and the current intersection count will be increased to reflect the number of intersections found at the closest *t* value. That is the full extent of the changes to EllipsoidIntersectScene with respect to using the spatial tree to query the polygon data as a broad phase.

The next change to the function is performed when testing the dynamic objects against the swept sphere. In our previous implementation we simply looped through every dynamic object and passed its buffers to the EllipsoidIntersectBuffers method. This essentially meant that every polygon in every dynamic object would have been run through the costly narrow phase. Now we will add a simple broad phase early-out test.

For each dynamic object, we transform its model space bounding box into world space using a new math utility function called TransformAABB (see MathUtility.cpp). This is a small function that takes an AABB and a transformation matrix and generates a new AABB that contains the passed AABB in its rotated/transformed form. We will look at the code to this function later.

```
    // Iterate through our triangle database
    DynamicObjectVector::iterator ObjIterator = m_DynamicObjects.begin();
    for ( ; ObjIterator != m_DynamicObjects.end(); ++ObjIterator )
    {
        DynamicObject * pObject = *ObjIterator;

        // Broad Phase AABB test against dynamic objects
        D3DXVECTOR3 Min = pObject->BoundsMin, Max = pObject->BoundsMax;

        MathUtility::TransformAABB( Min, Max, pObject->CollisionMatrix );
```

As we now have the current dynamic object's box in world space and we also stored the ellipsoid's world space bounding box extents in the m_vecEllipsoidMin and m_vecEllipsoidMax member variables at the start of the function, we can perform the AABB/AABB intersection test and skip all the dynamic object collision code that we discussed in the previous lesson. That is, if the boxes do not intersect, we skip to the next iteration of the loop and the next dynamic object to test. The remainder of the function is identical to the prior version.

```
        if ( !AABBIntersectAABB( Min,
                                 Max,
                                 m_vecEllipsoidMin,
                                 m_vecEllipsoidMax ) ) continue;

        // Calculate our adjustment vector in world space. This is for our
        // velocity adjustment for objects so we have to work in the original
        // world space.
        vecEndPoint = (Vec3VecScale( eCenter, Radius ) +
                       Vec3VecScale( eVelocity, Radius ));

        // Transform the end point
        D3DXVec3TransformCoord( &eAdjust, &vecEndPoint, &pObject->VelocityMatrix );

        // Translate back so we have the difference
```

```
        eAdjust -= vecEndPoint;

        // Scale back into ellipsoid space
        eAdjust  = Vec3VecScale( eAdjust, InvRadius );

        // Perform the ellipsoid intersect test against this object
        ULONG StartIntersection =
              EllipsoidIntersectBuffers( *pObject->pCollVertices,
                                        *pObject->pCollTriangles,
                                         eCenter,
                                         Radius,
                                         InvRadius,
                                         eVelocity - eAdjust,
                                         eInterval,
                                         Intersections,
                                         IntersectionCount,
                                         &pObject->CollisionMatrix );

        // Loop through the intersections returned
        for ( i = StartIntersection; i < IntersectionCount; ++i )
        {
            // Move us to the correct point (including the objects velocity)
            // if we were not embedded.
            if ( Intersections[i].Interval > 0 )
            {
                // Translate back
                Intersections[i].NewCenter      += eAdjust;
                Intersections[i].IntersectPoint += eAdjust;

            } // End if not embedded

            // Store object
            Intersections[i].pObject = pObject;

        } // Next Intersection

    } // Next Dynamic Object
```

Once each dynamic object has been tested and the results added to the Intersections array, we call EllipsoidIntersectBuffers one more time to perform the narrow phase tests on any geometry stored in the collision system's static geometry buffers. As discussed, now that we will be storing our static geometry in the spatial tree, these buffers will be empty when the collision system is being used by our applications. However, the logic is still supported for applications that may wish to use the collision library without having the implementation of a spatial tree forced upon them. Every polygon in these static geometry buffers is passed straight to the narrow phase, so there will be no broad phase implementation for geometry stored here. Since we have spatial trees at our disposal now, you will probably not want to store geometry in these buffers in future applications.

```
    // Perform the ellipsoid intersect test against our static scene
    EllipsoidIntersectBuffers( m_CollVertices,
                               m_CollTriangles,
                               eCenter,
                               Radius,
                               InvRadius,
```

```
                                    eVelocity,
                                    eInterval,
                                    Intersections,
                                    IntersectionCount );
```

As discussed in the previous lesson, if the caller requested that the intersection information be returned in world space (instead of ellipsoid space), we loop through each intersection that we have collected and transform the new position, collision normal, and intersection point into world space.

```
    // If we were requested to return the values in normal space
    // then we must take the values back out of ellipsoid space here
    if ( !bReturnEllipsoidSpace )
    {
        // For each intersection found
        for ( i = 0; i < IntersectionCount; ++i )
        {
            // Transform the new center position and intersection point
            Intersections[ i ].NewCenter =
                                Vec3VecScale( Intersections[i].NewCenter,
                                              Radius );

            Intersections[ i ].IntersectPoint =
                                Vec3VecScale( Intersections[ i ].IntersectPoint,
                                              Radius );

            // Transform the normal
            D3DXVECTOR3 Normal = Vec3VecScale( Intersections[ i ].IntersectNormal,
                                               InvRadius );

            D3DXVec3Normalize( &Normal, &Normal );

            // Store the transformed normal
            Intersections[ i ].IntersectNormal = Normal;

        } // Next Intersection

    } // End if !bReturnEllipsoidSpace

    // Return hit.
    return (IntersectionCount > 0);
}
```

As you can see, the changes to this core detection step method have been minimal. We will now discuss the new methods used by this function for intersection testing with the spatial tree.

CollectTreeLeaves – CCollision

As we saw in the previous function, if a spatial tree has been registered with the collision system, the CollectTreeLeaves method is called to fill the collision system's leaf array with all the leaves which intersect the world space bounding box of the swept ellipsoid. This function is extremely small. It first makes sure that the leaf list is emptied so that no leaf pointers exist in the leaf from a previous query.

Then this leaf list is passed into the spatial tree's CollectLeavesAABB method along with the world space AABB of the swept ellipsoid.

```
void CCollision::CollectTreeLeaves( )
{
    // Clear the previous leaf list
    m_TreeLeafList.clear();

    // Collect any leaves we're intersecting
    m_pSpatialTree->CollectLeavesAABB( m_TreeLeafList,
                                       m_vecEllipsoidMin,
                                       m_vecEllipsoidMax );
}
```

When this function returns program flow back to the EllipsoidIntersectScene method, the CCollision::m_TreeLeafList member will contain a list of pointers for all leaves that were intersecting the ellipsoid's AABB.

EllipsoidIntersectTree – CCollision

When the CollectTreeLeaves method returns back to EllipsoidIntersectScene, all the leaves have been collected and stored. Next a call is made to the EllipsoidIntersectTree method which iterates through every polygon in those leaves and performs the narrow phase test. Although this function code looks quite large, for the most part it is an exact duplicate of the EllipsoidIntersectBuffers method used to perform the narrow phase test on the other geometry pools. The reason that most of that code has been cut and pasted into this function is that we have to do some additional testing and are working with slightly different source data.

The ellipsoid's eSpace center position and velocity vector are passed in along with the ellipsoid radius and inverse radius vectors. We are also passed a float reference that is used to return the *t* value of intersection back to the caller. The function also requires an array of CollIntersect structures which it will populate when intersections are found that occur at a closer *t* value that the one passed in the eInterval parameter. The final parameter contains the number intersections currently in this array. As you have probably noticed, the parameter list is identical to EllipsoidIntersectBuffers, which we discussed in detail in the previous lesson. That is because this is essentially the same function.

The first thing we do is store the current number of intersections in the passed array in the FirstIndex local variable. This is so we know where we have started adding structures so that we can return this information back to the caller. We then store the current value of the application counter in a local variable so that we can use it to make sure that polygons that exist in multiple leaves do not get tested more than once.

```
ULONG CCollision::EllipsoidIntersectTree( const D3DXVECTOR3& eCenter,
                                          const D3DXVECTOR3& Radius,
                                          const D3DXVECTOR3& InvRadius,
                                          const D3DXVECTOR3& eVelocity,
                                          float& eInterval,
```

```
                                        CollIntersect Intersections[],
                                        ULONG & IntersectionCount )
{
    D3DXVECTOR3 ePoints[3], eNormal;
    D3DXVECTOR3 eIntersectNormal, eNewCenter;
    ULONG       NewIndex, FirstIndex, CurrentCounter;
    long        i, j;
    bool        AddToList;

    // FirstIndex tracks the first item to be added the intersection list.
    FirstIndex = IntersectionCount;

    // Extract the current application counter to ensure duplicate
    // polys are not tested multiple times in the same iteration / frame
    CurrentCounter = GetGameApp()->GetAppCounter();
```

Now we set up a loop to iterate through each collected leaf in the leaf list. For each leaf, we extract its polygon count and set up a loop to iterate though every polygon in that leaf. In this inner loop, we fetch a pointer to the current CPolygon we are going to test in the current leaf we are processing.

```
    // Iterate through our triangle database
    ISpatialTree::LeafList::iterator Iterator = m_TreeLeafList.begin();
    for ( ; Iterator != m_TreeLeafList.end(); ++Iterator )
    {
        ILeaf * pLeaf = *Iterator;
        if ( !pLeaf ) continue;

        // Loop through each polygon
        for ( i = 0; i < (signed)pLeaf->GetPolygonCount(); ++i )
        {
            // Get the polygon
            CPolygon * pPoly = pLeaf->GetPolygon( i );
            if ( !pPoly ) continue;
```

Next we test the polygon's m_nAppCounter member. If we find that it is the same as the current application counter value, it means we have already processed this polygon when visiting an earlier leaf in the list. Therefore, we skip any further testing on this polygon and go to the next polygon in the list.

```
            // Skip if this poly has already been processed on this iteration
            if ( pPoly->m_nAppCounter == CurrentCounter ) continue;
            pPoly->m_nAppCounter = CurrentCounter;
```

If we make it this far in the inner loop, it means we have not yet tested the current polygon. We now perform an additional broad phase step which is an AABB/AABB intersection test between the polygon's bounding box and the bounding box of the swept ellipsoid. If the bounding boxes do not intersect then the swept ellipsoid cannot possibly intersect the polygon, so we skip to the next iteration of the loop. With large leaf sizes, this minimizes the number of polygons passed to the narrow phase test and generates a considerable performance increase.

```
            // Are we roughly intersecting the polygon?
            if ( !AABBIntersectAABB( m_vecEllipsoidMin,
                                     m_vecEllipsoidMax,
                                     pPoly->m_vecBoundsMin,
```

```
                                        pPoly->m_vecBoundsMax ) ) continue;
```

If we make it this far, then we have a polygon that may well intersect the ellipsoid. Thus it will need to be passed on to the narrow phase.

Our next step would usually be to transform the vertices of the triangle into eSpace and then pass this triangle into the SphereIntersectTriangle method. However, our tree does not store triangles; it stores clockwise winding N-gons that describe triangle fans with an arbitrary number of triangles. For example, a polygon might be a clockwise winding of four vertices describing a quad comprised of two triangle primitives. Since our SphereIntersectTriangle method only works with triangles and not N-gons (as its name certainly suggests), we will need to set up an inner loop that will iterate though every triangle in the polygon and send it to the SphereIntersectTriangle method, one at a time. For example, in the case of a quad, we would need to call SphereIntersectTriangle twice, once for its first triangle and once for its second triangle.

To calculate how many triangles comprise a clockwise winding N-gon we simply subtract 2 from the number of vertices. Figure 14.92 demonstrates this relationship by showing a clockwise winding N-gon with 8 vertices and 6 triangles.

As you can see, we can find the vertices for each triangle by stepping around the vertices in a clockwise order. The first triangle will be formed by the first three vertices, and from that point on, we can just advance the last two vertices to form the next triangle. We can see for example that the first vertex V1 forms the first vertex for all triangles. The other two vertices are found for every triangle simply by stepping around the edges. For example, the last two vertices of the first triangle are v2 and v3. The last two vertices of the second triangle are v3 and v4, and so on.

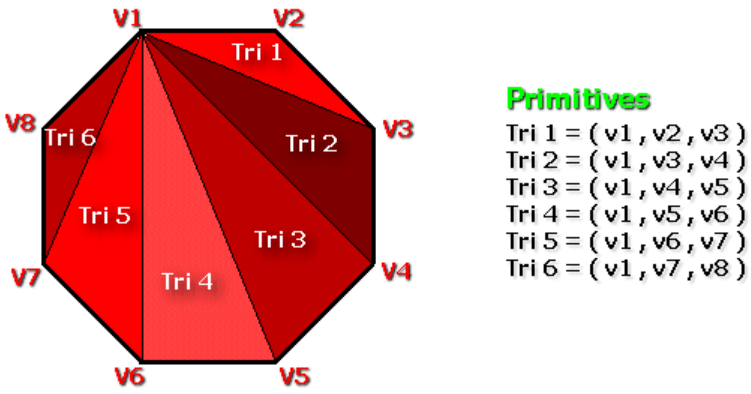


Figure 14.92

In the next section of code we start by transforming the polygon's normal into eSpace. We then set up a loop to iterate through every triangle in the polygon. In each iteration we select the three vertices that form the current triangle we are processing (see Figure 14.92) and transform them into eSpace before sending them into the SphereIntersectTriangle method.

```
            // Transform normal and normalize
            eNormal = Vec3VecScale( pPoly->m_vecNormal, Radius );
            D3DXVec3Normalize( &eNormal, &eNormal );

            // For each triangle
            for ( j = 0; j < pPoly->m_nVertexCount - 2; ++j )
            {
                // Get points and transform into ellipsoid space
                ePoints[0] = Vec3VecScale( (D3DXVECTOR3&)pPoly->m_pVertex[0],
```

```
                                    InvRadius );

            ePoints[1] = Vec3VecScale( (D3DXVECTOR3&)pPoly->m_pVertex[ j + 1 ],
                                    InvRadius );

            ePoints[2] = Vec3VecScale( (D3DXVECTOR3&)pPoly->m_pVertex[ j + 2 ],
                                    InvRadius );

            // Test for intersection with a unit sphere and the
            // ellipsoid space triangle

            if ( SphereIntersectTriangle( eCenter,
                                          1.0f,
                                          eVelocity,
                                          ePoints[0],
                                          ePoints[1],
                                          ePoints[2],
                                          eNormal,
                                          eInterval,
                                          eIntersectNormal ) )
            {
```

Note: The above code could be optimized by pre-transforming the vertices once, storing them in a local buffer, and then subsequently iterating through the triangles.

If the SphereIntersectTriangle method returns true then the sphere does intersect the triangle and eInterval will contain the *t* value of intersection. If eInterval is larger than zero then the intersection happened at some distance along the ray and we can calculate the new non-intersecting position of the sphere by adding the velocity vector scaled by the *t* value to the original centerpoint of the sphere. If eInterval is less than zero, then the sphere is embedded inside the triangle and eInterval describes the distance to move the sphere position along the direction of the plane normal in order to un-embed it.

```
                // Calculate our new sphere center at the point of intersection
                if ( eInterval > 0 )
                    eNewCenter = eCenter + (eVelocity * eInterval);
                else
                    eNewCenter = eCenter - (eIntersectNormal * eInterval);
```

The rest of the function is identical to the EllipsoidIntersectBuffers method. If there are currently no intersections stored in the intersection array, or if the *t* value for this intersection is less than the *t* value for the intersections currently stored in this array, we add the intersection information to the first element in the intersections array and set the intersection count to zero.

```
                // Where in the array should it go?
                AddToList = false;
                if ( IntersectionCount == 0 ||
                     eInterval < Intersections[0].Interval )
                {
                    // We either have nothing in the array yet, or the new
                    // intersection is closer to us
                    AddToList         = true;
                    NewIndex          = 0;
                    IntersectionCount = 1;
```

```
                    // Reset, we've cleared the list
                    FirstIndex        = 0;

                } // End if overwrite existing intersections
```

Alternatively, if eInterval is the same as the *t* value for the intersections stored in the Intersections array, then we add it to the end of the list and increase the intersection count member.

```
            else if ( fabsf( eInterval - Intersections[0].Interval )< 1e-5f )

              {
                    // It has the same interval as those in our list already,
                    // append to
                    // the end unless we've already reached our limit
                    if ( IntersectionCount < m_nMaxIntersections )
                    {
                        AddToList         = true;
                        NewIndex          = IntersectionCount;
                        IntersectionCount++;

                    } // End if we have room to store more

                } // End if the same interval
```

If we get to this point in the inner loop and find that the AddToList boolean has been set to true, it means the current triangle being tested was either closer or at the same distance along the ray as those intersections currently stored in the intersection array. In this case, we fill out the element in the array with the eSpace intersection information for the triangle.

```
                // Add to the list?
                if ( AddToList )
                {
                    Intersections[ NewIndex].Interval     = eInterval;

                    Intersections[ NewIndex].NewCenter    = eNewCenter +
                                                            (eIntersectNormal
                                                            * 1e-3f);

                    Intersections[ NewIndex].IntersectPoint = eNewCenter -
                                                             eIntersectNormal;

                    Intersections[ NewIndex].IntersectNormal= eIntersectNormal;
                    Intersections[ NewIndex].TriangleIndex   = 0;
                    Intersections[ NewIndex].pObject         = NULL;

                } // End if we are inserting in our list

            } // End if collided

        } // Next Triangle

    } // Next Polygon
```

```
    } // Next Leaf

    // Return hit.
    return FirstIndex;
}
```

We have now covered all the changes to the collision system. The end result is a powerful collision query engine that we can use in all future lab projects. The difference in query times with the spatial tree broad phase is quite significant. You can test this for yourself when you experiment with the lab project source code.

14.23 The TransformAABB Math Utility Function

In the EllipsoidIntersectScene method we saw a call being made to the MathUtility::TransformAABB method. This call transformed the AABB of a dynamic object into world space. The method was passed the minimum and maximum extents of a bounding box and a transformation matrix with which to transform the AABB.
Let us be clear about exactly what this function is supposed to do. It does not actually transform the box in the traditional sense, as an AABB by its very nature is always aligned with the world axes. If we were to rotate the box we would in fact have an OBB (oriented bounding box). The OBB would have to be stored in a very different way (position, extents, and orientation of its local axes) and this is not our goal. OBBs are much trickier to work with, and generally more expensive (although tighter fitting), and since our collision system is currently working with AABBs throughout, we will continue to do that.

What this function will do however is calculate a new AABB that contains the passed AABB post-rotation. To understand this concept, take a look at the AABB shown in Figure 14.93. In this example the box is assumed to be centered at the origin of the coordinate system with a maximum extents position vector of (10,10).

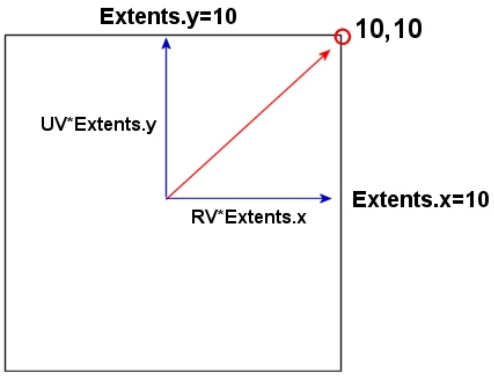


Figure 14.93

We know that an extents vector of <10,10> means the box expands to a maximum of 10 units along the world x axis and a maximum of 10 units along the world y axis. That is, we know that in the case of an AABB, Extents.x describes the position on the world X axis where the right side of the box intersects it and Extents.y describes the position along the world Y axis where the top of the box intersects it. For ease explanation we will deal with a 2D coordinate system, but the same holds true in 3D.

We know that in the case of an axis aligned bounding box, its local right and up vectors are aligned with the world X and Y axes respectively. However, if we perceive the world axes to be the local right and up vectors for the box, we can see that Extents.x describes the amount we would need to scale the box's unit length right vector (RV) to create a vector that when added to the box centerpoint describes a point

on the plane of its right face. Likewise, the unit length up vector for the box (UV), when scaled by the y component of the Extents vector and added to the centerpoint of the box, describes a point on the plane on which its top face lies. We can see that scaling the right vector and the up vector of the AABB by the components of the extents vector essentially just returns the extents vector (just as if we were transforming the extents vector by an identity matrix). However, what we want to do is transform the up and right vectors of the AABB by the passed matrix and calculate a new AABB that encompasses the OBB these new vectors describe.

In Figure 14.94 we show what the box would look like if we were to rotate these vectors 45 degrees to the left, simulating what would happen if the passed transformation matrix contained a 45 degree rotation. We can see that the rotated right and up vectors (which were scaled by the extents vectors components) still describe the location of the right and top planes of the box faces, but they are no longer aligned to the world axes. Essentially, these vectors and the center point describe an OBB now.

We know that if we rotate an AABB 45 degrees to the left or right it will pierce the original extents and a new larger AABB will need to be calculated to contain it. It is this new AABB (the red dashed box in Figure 14.93) that we want this function to efficiently calculate.

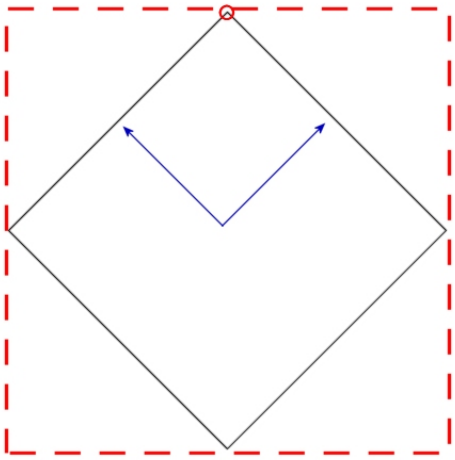


Figure 14.94

The first section of code generates the scaled and rotated right, up, and look vectors (3D) for the box.

First we calculate the center of the box and subtract it from the passed maximum extents vector. This gives us the diagonal half length of the box shown as the red arrow in Figure 14.93. We then make a copy of the passed matrix into a local variable and zero out its translation vector (we will deal with translation separately at the end of the function).

```
void MathUtility::TransformAABB( D3DXVECTOR3 & Min,
                                 D3DXVECTOR3 & Max,
                                 const D3DXMATRIX & mtxTransform )
{
    D3DXVECTOR3 BoundsCentre = (Min + Max) / 2.0f;
    D3DXVECTOR3 Extents = Max - BoundsCentre, Ex, Ey, Ez;
    D3DXMATRIX  mtx = mtxTransform;

    // Clear translation
    mtx._41 = 0; mtx._42 = 0; mtx._43 = 0;
```

Our next step is to transform the centerpoint of the box by the transformation matrix which rotates the centerpoint into the space described by the passed matrix.

```
    // Compute new centre
    D3DXVec3TransformCoord( &BoundsCentre, &BoundsCentre, &mtx );
```

Now we need to compute the scaled and rotated local axes for the box shown as the rotated blue arrows in Figure 14.94. We said that we needed to scale the unit length axes by their matching components in the extents vector so that the local up, right, and look vectors are grown in length such that they describe distances to the planes of the top, right and front faces. However, as the passed transformation matrix already contains the rotated (but unit length) right, up, and look vectors in its 1st, 2nd, and 3rd rows respectively, we can just scale the rotated unit length axes by their matching components in the extents vector to get the rotated axes:

```
// Compute new extents values
Ex = D3DXVECTOR3( mtx._11, mtx._12, mtx._13 ) * Extents.x;
Ey = D3DXVECTOR3( mtx._21, mtx._22, mtx._23 ) * Extents.y;
Ez = D3DXVECTOR3( mtx._31, mtx._32, mtx._33 ) * Extents.z;
```

Let us plug in some example values to see what has happened so far. We will once again use the 2D case for our diagrams.

Imagine we pass in a transformation matrix that will rotate our original 2D box (with a half diagonal length vector of <10, 10>) 45 degrees to the left. We know in such a case that the first row of the matrix which contains the rotated right vector will store the vector <0.707, 0.707> and the up vector stored in the matrix will be <-0.707, 0.707>. These describe the unit length vectors <1,0,0> and <0,1,0> rotated 45 degrees to the left:

RV = [0.707 , 0.707]
UV = [-0.707, 0.707]

These vectors current describe the directions we would need to travel from the box centerpoint to reach the rotated right and top faces of the box. They do not yet describe the distance over which to travel. However, in the above code we scale the up and right vectors (and the look vector in our 3D code) by their matching components in the Extents vector to generate the up and right vectors shown in Figure 14.95. These describe the direction and distance to the right and top faces of the rotated AABB (OBB).

In Figure 14.95 we see that Ex and Ey contain the results of the scaled RV and UV vectors (scaled by the original extent vector components – which are both 10 units). When added to the center position of the box, they describe points in the center of the right and top faces.

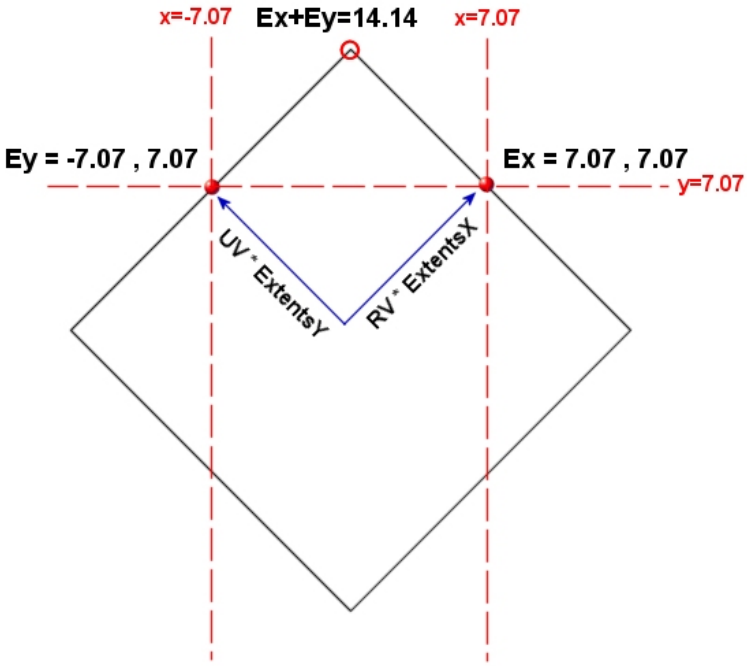


Figure 14.95

Ex = RV*10 = [7.07 , 7.07]
Ey = UV*10 = [-7.07 ,7.07]

So with these two points, how do we calculate the new extents of the rotated box along the world X and Y axes so that we can build a new AABB that encompasses it?

It just so happens that summing the absolute x components of each vector will tell us the size of the box along the world X axis and summing the y components of both vectors will tell us the height of the rotated box in world aligned space. For example:

NewExtents.x = abs(Ex.x) + abs(Ey.x) = 7.07 + 7.07 = 14.14
NewExtents.y = abs(Ex.y) + abs(Ey.y) = 7.07 + 7.07 = 14.14

As you can see, by summing the like components of the scaled and rotated vectors, we can compute exactly how much of the world X and Y axes this rotated OBB covers. NewExtents.x now describes the distance from the center of the box to the tip of the diamond on the right hand side of the rotated box. NewExtents.y describes the distance from the center of the box to the tip of the diamond at the top of the OBB (see Figure 14.96).

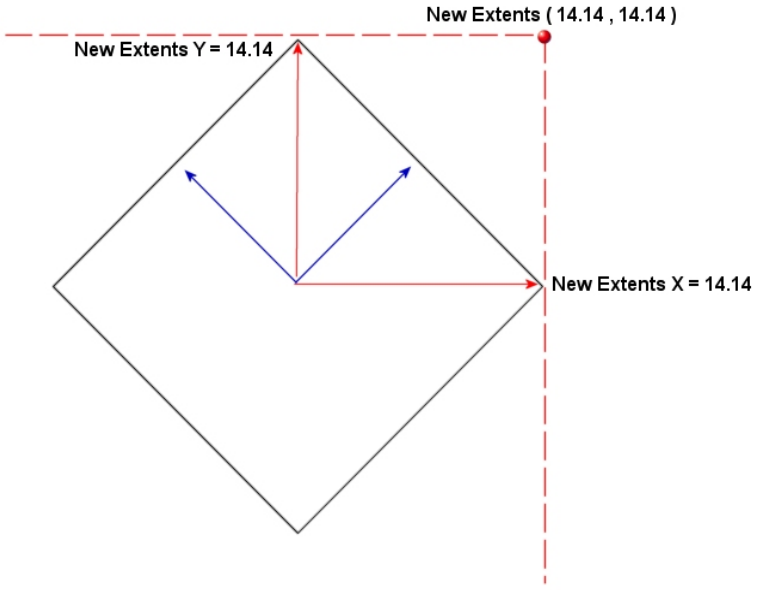


Figure 14.96

Looking at figure 14.96 we can see that the new diagonal half length of the box is therefore:

NewMaxExtents = [New Extents.x , New Extents.y] = [14.14 , 14.14]

Let us now take a look at the rest of the code to the function. In the previous section we calculated the rotated and scaled up, right, and look vectors of the OBB and stored them in the vectors Ex, Ey and Ez (the blue arrows in Figure 14.96). As we just discovered, all we have to do is sum the absolute values of the like components in each vector to get the new extents vector for the box.

```
    // Calculate new extents actual
    Extents.x = fabsf(Ex.x) + fabsf(Ey.x) + fabsf(Ez.x);
    Extents.y = fabsf(Ex.y) + fabsf(Ey.y) + fabsf(Ez.y);
    Extents.z = fabsf(Ex.z) + fabsf(Ey.z) + fabsf(Ez.z);
```

At this point, Extents holds the diagonal half length vector of the AABB we wish to create (which will bound the rotated original AABB). We can calculate the minimum extent position vector for the box by subtracting the new extents vector (diagonal half length vector) from the box center, and calculate the new maximum extents position vector by adding it to the box center. Finally, we also add the minimum and maximum extents vectors of the new AABB to the translation vector stored in the fourth row of the passed matrix. This means the new AABB is not only recalculated to encompass the rotation applied to the original AABB, but is also translated into the position described by the passed matrix.

```
    // Calculate final bounding box (add on translation)
    Min.x = (BoundsCentre.x - Extents.x) + mtxTransform._41;
    Min.y = (BoundsCentre.y - Extents.y) + mtxTransform._42;
    Min.z = (BoundsCentre.z - Extents.z) + mtxTransform._43;
    Max.x = (BoundsCentre.x + Extents.x) + mtxTransform._41;
    Max.y = (BoundsCentre.y + Extents.y) + mtxTransform._42;
    Max.z = (BoundsCentre.z + Extents.z) + mtxTransform._43;
}
```

And there we have it. This is a very handy way to calculate a new AABB for an AABB that you wish to transform into another space. The result is an AABB which completely contains the original rotated AABB. Keep in mind that this AABB has the potential to be much larger than its untransformed predecessor, which means that query accuracy can be decreased as a result of the looser fitting volume.

Conclusion

In Lab Project 14.1 you will see a lot of code associated with tree rendering which we have not yet covered. That is because this lab project will span two lessons, with the render system being explained in the next one. For now, you should focus your attention on the main implementation of our tree classes and the changes made to the collision system that allow us to use the spatial tree as a broad phase.

In the next lesson we will see the same spatial tree being used to speed up rendering. This will lay the foundation for a rendering system that will be utilized by our spatial trees and by the powerful BSP/PVS rendering engine we will create as we wrap up this course.