Chapter Twelve

Skinning II

Introduction

At this point in the course we have learned how to load X file frame hierarchies and traverse and render them. We have also learned how to load skinned mesh data from an X file and render those skinned meshes such that the hierarchy acts as a system of bones that influence the world space positions of the skin's vertices. Of course, loading data from an X file is only half the story. We can also procedurally generate meshes, hierarchies, bones, and skins. This can prove to be handy for lots of things, like random generation of terrain or even for vegetation on that terrain to cite just two obvious examples. Typically such code uses algorithms that generate geometry based on both fixed rules and some degree of randomness. How the two concepts work together is generally based on some set of input parameters and controls to the procedure. For example, if we were randomly generating a terrain mesh, we might specify some rules to the procedure stating that water can only exist below a certain altitude, or that snow can only be placed on mountaintops above a certain altitude. In both cases, those altitudes are likely to be input to the terrain generator via a configuration file or through a GUI tool.

Figure 12.1

Regardless of the example we choose, the important point is that you will undoubtedly reach a point at some time in your programming career where you will have to generate meshes and/or texturing information from within your code. For some game assets it is just easier to do it this way (versus getting your artists to create everything by hand). Trees are a good example. Every tree is different in reality, so if you wanted to represent even a small forest of say, 40 unique trees, it would require the artist to generate 40 different tree meshes, where each branch and maybe even each leaf would have to be carefully hand-crafted and textured. The time involved would be considerable, and most serious game development projects would simply not have the resources to spare for such an undertaking. So not surprisingly, representing realistic trees within a game world has typically been an area that has fallen to tools developers to aid the creative process. There are currently some commercial products (e.g., SpeedTree™) available that are designed to do nothing more than generate realistic trees that look organic while remaining within a specified polygon budget (the polygon count of a single tree is very important as we rarely have just one tree in a given scene – especially an outdoor scene). These products produce some incredibly realistic results, but they can also be very expensive. Since commercial tools are generally not in the budget for most students, we decided to introduce code for generating our own trees. This will be a great way to conclude our studies of skins and skeletons and it has the added benefit of providing you with a useful tool in the short term at no extra cost (other than the time you will have to spend writing the code). To be clear, we are not going to claim that our trees are going to exhibit the same level of quality or performance as those produced in the commercial tools, but we are sure that you will find them to be an adequate substitute in the short term while you are still in training.

In Figure 12.1 we see a tree that has been generated procedurally using a new class that we will create in this chapter (called CTreeActor). What cannot be seen in the image is that this tree is really a combination of skinned meshes and bones. The bones are animated to simulate how a tree might move in blowing wind. Our tree class will be derived from CActor and as such we can reuse much of its functionality. CActor already has code to represent, animate, and render skinned meshes, so this is code we will not have to rewrite in our tree class (thankfully). In fact, CTreeActor will simply add some hierarchy and skinned mesh creation functions to the base class that allow us to populate the actor's hierarchy using a different means. As we have seen, currently CActor can be populated only by loading an X file. With CTreeActor we will add functions that generate the bone and mesh data for tree meshes and add them to the hierarchy manually.

> **Note**: While it is perfectly legitimate to use CTreeActor to generate trees in your code at level load time (for example), it is really supposed to be used as tool. That is, usually you will use CTreeActor to generate some number of random trees and when it creates a tree you like, you will save that actor to an X file for use in your game. This way, your actual game code need only load the trees using a normal CActor just like any other X file. You may also wish to save the trees out to X files and then import them into a level editor for placement. GILES™ now ships with a tree plug-in to allow you to create trees from within its GUI. The GILES™ tree plug-in uses CTreeActor for its tree generation.

A CTreeActor object will represent a single tree in the scene. Its frame hierarchy will describe the bones of the entire tree representation (the skeleton of the trunk and all its branches). CTreeActor's creation functions will also generate animation data for this hierarchy programmatically that will animate the tree such that its branches and leaves blow from side to side in simulated wind. Sharing this same bone hierarchy will be multiple meshes, one skinned mesh for each branch. What we will have at the end of this chapter (and associated lab project) is a class that we can use to generate tree meshes with random configurations with a single function call. For the most part, using the tree class will be no more complicated than calling the CTreeActor::GenerateTree function where we would usually call the CActor::LoadActorFromX function. Therefore, the code that we have added to CTreeActor is strictly confined to the creation process as a replacement for loading of X file data. Once the CTreeActor::GenerateTree function has built the frame hierarchy and its various branch meshes, we can position, animate, and render the tree using the same CActor methods we have always used.

Since this is going to be a fairly complex task, we will break the coverage of CTreeActor into two separate lab projects (12.1 and 12.2) which will be covered in two different sections in this textbook. In the first section of this book we will write the code that generates the main branch structure of the tree (Lab Project 12.1). This will involve procedurally creating multiple skinned meshes (one per branch). Once we have our CTreeActor fully capable of creating such a network of animating skinned branch meshes, we will add simple leaves to our trees (Lab Project 12.2). Doing it in two steps like this will make the code easier to understand and the entire process a little less overwhelming.

> **Note:** This chapter obviously represents a departure from the way we have done things thus far in the training series. Generally, source code discussions were limited to our workbooks while the textbooks concentrated more on higher level theoretical ideas. So it is worth noting that the line between textbooks and workbooks is likely to blur quite a bit as we move forward in this course (and indeed in the remainder of the series). As we start to get into more implementation specific topics (such as we will here), it will become difficult and impractical to split things up as cleanly as we have done in the past. Instead what we will find is that the work will be shared between the two books. For example, there will be times when the textbook will cover the implementation details in addition to the design ideas and

theory, while the workbook focuses on how to integrate and use the systems developed in the textbook. To be sure, this will not happen all the time, and again, the line will be somewhat blurred. Certainly there will be times in the future when the old model shows itself again and the code will be confined almost exclusively to the workbooks. But when it is not practical to do so, we will approach the topics like you will see here in this chapter. Indeed, throughout the rest of this course, you should expect to see this approach being used in nearly every chapter.

Also note that since this chapter is essentially just an extension of the ideas introduced in the last chapter, there will be no new accompanying presentation. We are going to focus exclusively on writing code this time around. Of course, the material in Lecture 11 is completely applicable to what we will do here, so please refer to the presentation as the need arises.

Figure 12.2

The tree in Figure 12.1 essentially consists of two geometry building techniques. Skinned meshes are used to represent the trunk and branches of the tree, while the leaves are represented using a series of criss-crossed quads. As we will be discussing the technique used for leaf generation in a later section, in this first section we will focus on creating only the trunk and branches of the tree. In essence, our first version of CTreeActor will only generate "winter" trees. Figure 12.2 shows what the tree shown in Figure 12.1 will look like without the leaves. This is what the trees we generate in the first section of this chapter will look like.

Of course, while it is essential to have trees at your disposal if you ever intend to represent an outdoor environment other than the Sahara Desert or the Atlantic Ocean, providing you with organic looking trees was not the motivation behind this lesson. The real goal behind this project was to come up with a task that would put to the test everything you have learned about since the beginning of the course, but with some new twists. For example, you learned how keyframe data is generated back in Chapter Ten, but so far we have only demonstrated how to load and playback that data. Similarly, in Chapter Eleven, we discussed the relationship between vertices in a skin, the bones that influence them, and the individual animations that animate those bones in the hierarchy. But even that material was focused on working with pre-generated data. In short, to really understand a skeletal animation system (or arguably, any system for that matter), it will be helpful to work on a project where every bit of data is generated from scratch right in your own code. This is exactly the educational experience that CTreeActor will provide, as you can see just by glancing at our to-do list:

- We will need to generate the meshes for each branch and of course, the bones which those meshes attach to.

- We will have to calculate the bone offset matrix for each bone in our hierarchy such that it is relative to the bone at which the branch mesh begins and not the root of the entire tree.

- We will have to manually connect the vertices to the bones that animate them.

- We will have to create an animation controller for the actor, which we will then populate with keyframe data that we will generate.

It seems that we have quite a bit of work to do, and indeed we do. But believe it or not, the actual code we will write to extend CActor into our derived tree class will be fairly small. It is still quite a complex process that we will be undertaking however, which is why we will first discuss the system design and try to nail down how it will all work. Then we can start dealing with source code and implementation details.

12.1 An Overview of the Tree Generation System

Our tree will be made up of a number of branches that will be randomly generated using a recursive process. Even the trunk of the tree will just be a branch (considered the "root branch" of the tree). Each branch will be "grown" using a recursive and random process that may, at any given segment within the branch, cause another child branch (or multiple child branches) to be generated. Each child branch generated from the root branch (the trunk) will also be built in exactly the same way. As each child branch is grown, there is a chance that any given segment within that branch may spawn multiple child branches of its own, and so on. Do not worry too much about what a branch segment is at the moment, we will get to that in a moment.

We will feed the generator procedure a number of input parameters that will be used to influence the growth of the tree and its individual branches. For example, one parameter we will feed in will be a growth direction vector that will describe the world space direction in which we wish the tree (the trunk branch) to initially start growing. We will also feed in probability variables that describe the odds that a branch segment might split into one, two, three or four more child branches at any given segment along its length. In addition we will feed in variables that influence the amount that a child branch's direction vector can be randomly deviated from its parent branch (while still maintaining the general overall growth direction of the tree as a whole). Further, we provide variables that allow us to specify the size of the tree and the resolution at which we place bones within that tree. Other variables will specify the resolution of the mesh branches, allowing us to control exactly how many vertices are used to create a given branch segment. This will allow us to directly influence the face count of the tree and tailor the tree to our specific polygon budget. So before we discuss any code, let us first examine the system that we will use at a high level. This way when we discuss the code later on, we will understand the input variables and how we are using them.

Figure 12.3 : A branch made out of a single cylinder

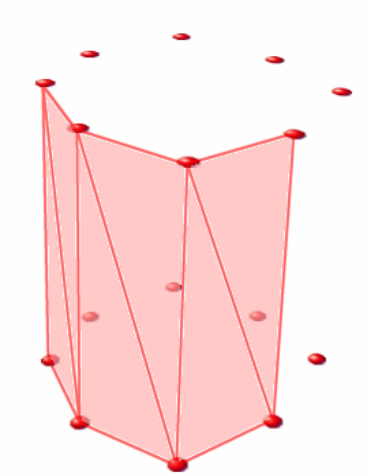

Figure 12.4: The two rows of vertices created triangles that wrap around a cylinder

As mentioned, our tree will essentially just be a hierarchy of branches. If we think in terms of primitive objects, a branch could be represented at a very coarse level by a cylinder mesh (see Figure 12.3). Already, this image looks like a very rough approximation of the trunk of a tree, especially when it is mapped using a tree bark texture. What we are looking at in Figure 12.3 is a single cylinder mesh. It has eight vertices arranged in a circle around the top and eight vertices arranged in a circle around the bottom of the cylinder. In this image we can only see four of the eight vertices used at the top and the bottom of the cylinder as we are only looking at one of its sides. As Figure 12.4 clearly demonstrates, we can use these two rows of vertices to create triangles that form the cylinder primitive. Each pair of vertices from the top and bottom rows can be indexed to form a quad (two triangles). When we do this for every matching pair of vertices in the top and bottom rows, we have an un-capped cylinder, which when textured, would look like the branch shown in Figure 12.3.

Having vertices only at the top and the bottom of each branch (one cylinder), does not afford us much flexibility when it comes to shaping the branch into something more interesting. While we could decrease the radius of the circle formed by the top row of vertices to make the cylinder get thinner as it nears its end, this would still be a perfectly uniform thinning out of the branch from bottom to top. We might imagine for example, that we would want the branch to end in a tip, possibly by moving all vertices in the top row into a single center point. However, the branch would then become a perfect cone and we really cannot have a tree where all the branches are perfectly coned shaped. So instead of making each branch using a single cylinder, a branch will be comprised of multiple cylinders stacked on top of each other.

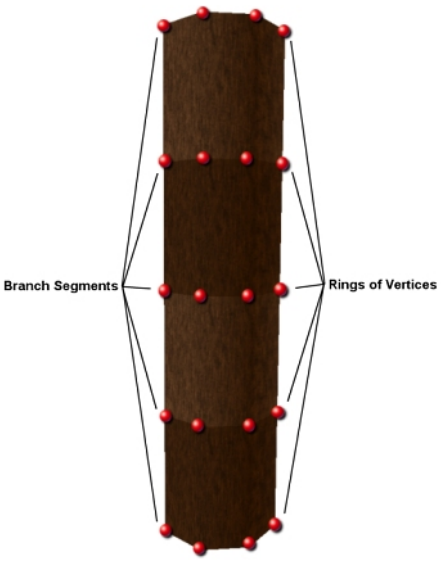

Figure 12.5 : A branch mesh constructed from multiple branch segments

We will refer to each cylinder as a **branch segment** as shown in Figure 12.5. Our branch will now consist of not just two rows of vertices arranged in a circular pattern, but *N* rows of vertices which form multiple branch segments. With the exception of the first row of vertices (at the bottom of the branch) and the last row of vertices (at the top of the branch), all other rows will consist of shared vertices that form the top of one branch segment and the bottom of another. In Figure 12.5, we show how our branch mesh might look consisting of multiple branch segments. All segments still belong to the same mesh, after all, they are all part of the same branch and we have already mentioned the fact that each branch will be its own skinned mesh.

Now that we have vertices at regular intervals along the branch, we can adjust the radius of the circles formed by each row of vertices such that the faces that form the branch get thinner and thicker as we desire. In our code, we will use the index of the branch segment within the branch we are currently building to determine the radius of its vertices (with some degree of randomness thrown in) such that all branches generally get thinner until they end in a point at their tip. It would be a waste representing the tip of any branch using a row of eight vertices that essentially exist in the same position, so when adding the final row of vertices to a branch (to complete the final segment of a branch), we will insert just a single vertex that exists at the center of the circle. Every vertex in the penultimate row of vertices will be indexed along with this final vertex to turn the final branch segment into a cone.

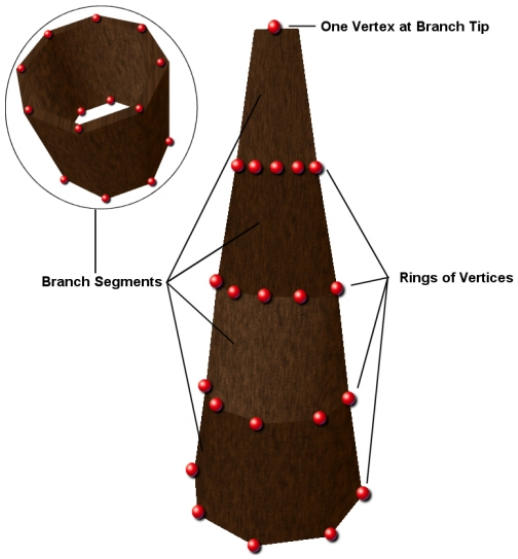

Figure 12.6

Figure 12.6 shows how a branch might look with uniform scaling applied to the vertices at each segment boundary. It also demonstrates how the final branch segment of any branch is terminated using a single vertex at its tip. While this shows a uniform scaling making the branch appear cone shaped, this is not something we are limited to doing. By changing the radius of the circle formed by each ring of vertices we can make the branch mesh swell or contract at the segment boundaries. The circular inset in figure

12.6 also reminds us that all branch segments (except the top segment) are essentially uncapped cylinders that slot together as a natural product of the fact that they share the same vertices at the segment boundaries. The top segment is obviously an exception to this rule where the faces of this segment will each be generated using the final 'tip' vertex and each pair of vertices from its base row of vertices.

So, each branch will be a mesh made up from a number of branch segments. Furthermore, each tree will be made from a number of branch meshes. In Figure 12.7 we see a tree that is made up of 7 branches. Admittedly it is a poor looking tree and there is much left to be done, but the relationship shown here is the important concept to understand.

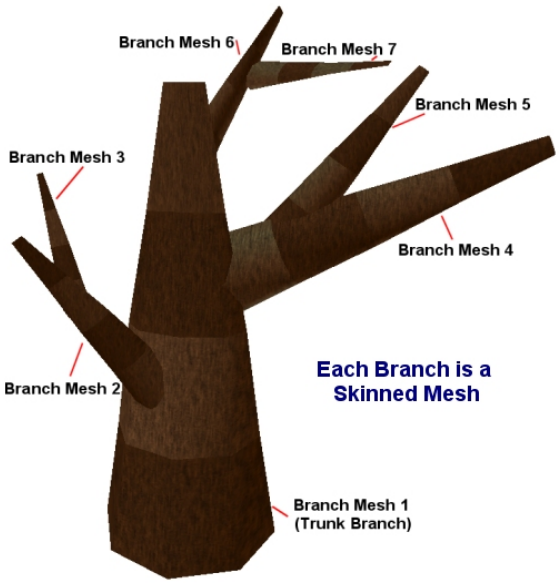

Figure 12.7

When we grow the tree, we initially start out with the root branch; more specifically, the root segment of the branch. We build the branch up segment by segment, at each step determining how big or wide that segment should be. We will also make a choice at each branch segment whether a new child branch (or multiple child branches) should be spawned from that segment. We can see in Figure 12.7, that at segment 2 of the trunk branch, a child branch was spawned off to the left, which itself set in motion a recursive procedure of building the child branch segment by segment in the same way. In this child branch we can see that while generating its third segment, another (smaller) child branch was spawned. After the recursive process ends for the child branches, the flow once again returns to the root branch, which is still half way through being built. On the third segment of the root branch it is determined again that another child branch should be spawned (this time off to the right), which itself spawns its own child branch during the construction of its second segment. When flow returns back to the root branch, and its fourth segment is added, the decision to split into a child branch is made yet again. This child also splits off into a separate child branch some way along its length.

We can definitely see the hierarchical pattern here. If we think about how we might represent this concept in a hierarchy (much like our frame hierarchy), in a way that is totally abstracted from any type of mesh data, we can think of the root segment of the trunk branch as being the root node in the hierarchy. The root node would have a child that would be the second segment in the root branch. This second level child would also have a child which would represent the third segment of the trunk branch, and the third segment would have a child that pointed at the data for the fourth and final branch segment in the trunk. So, each individual branch would be represented in the hierarchy as a list of N nodes arranged in a strict parent-child relationship that represented the N segments of that branch.

Now let us think about how the child branches could be stored. The first child node of the trunk branch in the hierarchy (which represents the second segment of the trunk) could have a sibling that is the first segment in the first child branch (the one that branches to the left). After all, the second segment of the trunk and the first segment of the child branch exist at the same level in the tree and should also exist at

the same level in the hierarchy. We then recur with that sibling pointer, such that each of its segments would be added in a parent-child relationship. So segment N of any branch is the parent of segment N+1 in the same branch.

In Figure 12.8, we see how this hierarchy might look in memory. To reduce clutter, we have only fully shown the trunk branch and the first child branch (protruding from the second segment of the trunk out to the left) in their entirety. When studying this image, do not concern yourself with meshes or bones for now; we are simply constructing a 'virtual tree' tree at this stage, where each branch segment has its own node in the hierarchy and the first segment in any child branch is connected as a sibling of the parent branch segment from which it was spawned.

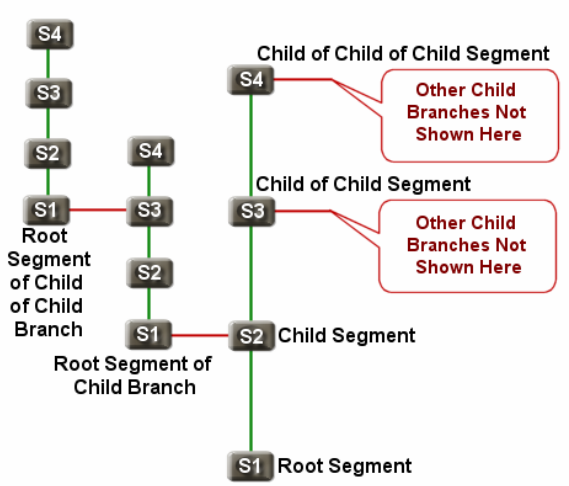

Figure 12.8

In Figure 12.8 we have inverted the diagram so that it reads bottom up. While this is not the typical way to draw a hierarchy, it corresponds better visually with the tree we saw in Figure 12.7. At the bottom of the image we see the root node of the hierarchy. This represents the first segment of the trunk branch. The parent/child relationship is depicted by vertical lines connecting the nodes, while the horizontal lines depict where a sibling relationship exists between two nodes. We can see that the root segment actually has two children arranged in a sibling list. The first child (S2) represents the second segment of the trunk branch, while a horizontal line connects node S2 to the root segment of a child branch. S2 of the root branch also has a child S3. S3 would also have a sibling but that is not shown here. The S3 node of the root branch has a child also (S4), which once again, would also have a sibling to the start of a child branch which is not shown here.

If we backtrack to node S2 of the root branch, we can see that it is in a sibling list with node S1 of a child branch. This tells us that at node S2 in the root branch, a new child branch was spawned. Node S1 in the child branch has one child (S2), but node S2 of the child branch has two children arranged in a sibling list. It points to node S3 (the third segment) in the child branch and node S1 in a new child that is spawned from this child branch at this third segment.

Study the diagram and make sure you understand the relationships as this will all be very important moving forward. If our tree was a single branch with four segments for example, this would create a four level deep hierarchy where each child (except the root) would be a child of the previous node. Each

level in the hierarchy would contain a single node. Any child branches that are spawned at a given branch segment, are arranged in a sibling list with that parent branch's segment node.

It is very important that you consider the node hierarchy shown in Figure 12.8 abstracted from the concept of meshes, vertices, or bones. We are currently just generating a hierarchy of data structures (nodes) that will describe to us the shape of the tree we need to generate. We might consider each node in the tree containing such information as the tree space position of that node and the direction it is facing. In fact, this is exactly how we will build our tree. We will essentially do it in two phases.

In the first phase, we will grow (node by node) a hierarchy of data structures describing the virtual tree. Then, once the shape of the virtual tree has been created, we will traverse this node hierarchy in a second phase and use the information to build the meshes and add the bones. This approach is preferable because it allows us to abstract the shape of the tree that we generate from the procedure used to skin it. After we have built the node hierarchy, we will traverse it in a second phase, and insert a ring of vertices at each node adding another segment to the branch. Of course, at a later time, you could decide to build the mesh data in a completely different way still using the same virtual tree information. For example, you might decide to use the node information to build the tree out of curved surfaces or even decide to drop the resolution such that a ring of vertices is only inserted every three nodes. These are just examples, but hopefully you understand why it is helpful to abstract the virtual tree generation process from the process that turns that virtual representation into a discrete polygonal representation.

As stated above, in our code, when we build the mesh, we will insert a ring of vertices at every node we generated in the virtual tree creation process. This means, with the exception of the first node of every branch, every other node we encounter will cause another branch segment (cylinder) to be generated. Obviously, the first node in a branch cannot possibly add another segment by itself as we need two rings of vertices to create a cylinder. Therefore, it is not until the second node is encountered and another ring of vertices is inserted that the first segment of the branch is complete. The ring of vertices we add at the second node however, also forms the bottom row of vertices for the next branch segment. So when we encounter the third node and add a third ring of vertices to the branch, this (along with the vertices from the second node) forms the second branch segment of the current branch being built.

Thus, our recursive procedure will initially grow a virtual tree out of a hierarchy of nodes, where each node contains data about that node. What information would we need to store in each node? First we will need to store the position of the node in tree space. In tree space, the root node of the trunk branch would be positioned at (0, 0, 0) in the coordinate system. However, what might not seem obvious at first is that in order to grow this node and spawn child nodes, we will need each node to also have a direction vector. This direction vector is a vector pointing in the direction of where the next node will be placed. More importantly, the direction vector can be thought of as a normal to a plane that passes through that node. When we generate the actual mesh for the branch, this is the plane upon which we will place the ring of vertices generated by that node.

To better understand the need to store the direction of a node, let us first just consider a four segment branch that forms the trunk of a tree growing directly upwards. In this case, the direction vector of each branch would simply point straight up to the next node (0,1,0). When we generate the mesh for the vertical tree, this direction vector also describes the normal to the plane on which the node is sitting. In this instance, each node would sit on an XZ aligned plane offset some distance vertically from the origin

of the coordinate system (except the first node which would exist on the XZ plane). As Figure 12.9 demonstrates, when we generate the mesh for a branch, the direction vector describes the normal of a plane. The node is assumed to be positioned at the center of a circle defined on that plane. Using the plane normal we can also generate two tangent vectors (vectors that lay on the plane forming an orthogonal axis with the normal vector). Essentially, once we have the three axes of the node, we can generate the N vertices at the node position and then push them out along the tangent vectors by some radius to form a circle of vertices that lay on the node's plane.

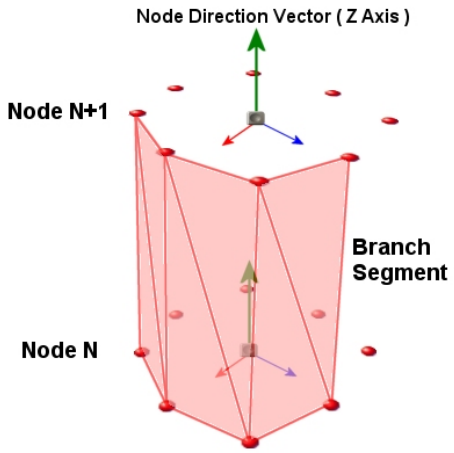

Figure 12.9

You might be wondering why each node would need its own direction vector if the tree is going to grow straight up most of the time anyway? Well, let us consider child branches that are spawned from a node of the trunk branch. They will almost certainly need to have their direction vectors deviated from the parent node's direction vectors, otherwise, all branches spawned would grow vertically straight up inside the trunk of the tree (we would not even see them).

One of the major aspects of the virtual tree generation process will involve deviation of a child's direction vector from that of its parent. For example, let us assume we start at the root node and assign it an initial 'straight up' direction vector of (0, 1, 0). Imagine that we continue to add child nodes with the same direction vector to the same branch, but then we hit a node where a random calculation says we need to add a new child branch. We know that this child branch must not share the same direction vector as the parent, so we will take the parent node's direction vector and rotate it (deviate it) by some amount. The maximum and minimum angles by which we deviate the child node's direction vector from its parent will be controlled by input parameters to the system. We will essentially use two angles to define a deviation cone. When a child branch is spawned and a new node is added to the parent node's child list, its direction vector will be generated by randomly deviating the parent direction vector within a specified range. As we have control over the deviation range via its input parameters, we can use fairly small ranges such that the child branches, while deviating from the parent direction, still follow the overall growth direction of the tree (appropriate for most trees). Alternatively, by specifying a larger random deviation range, we can have child branches growing off at wild angles and even coming back down against the initial growth direction of the root branch (useful for modeling some tree types).

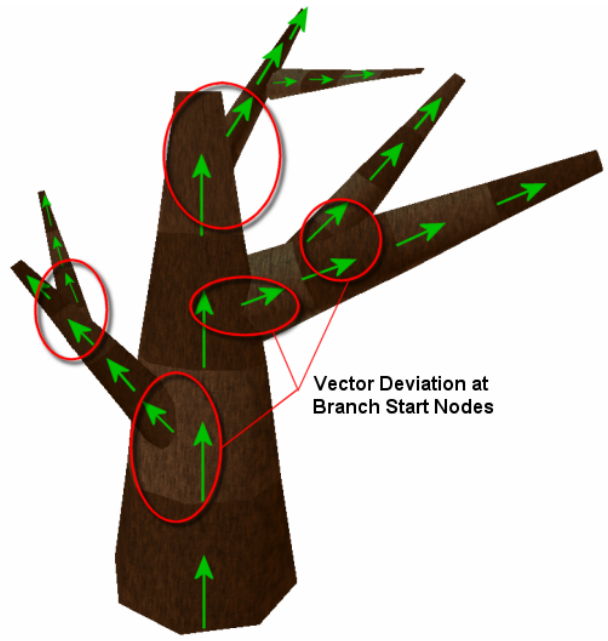

Figure 12.10 : Deviated Direction Vectors

Do not worry about how we deviate a vector within certain restraints as we will get to that in a moment. Figure 12.10 clearly shows why the nodes of each branch will need to have deviated direction vectors from that of its parent branch segments.

Looking at the tree in Figure 12.10, we see that it still is not quite there yet. While the branches shoot out at their own directions, the individual segments within a given branch are all still far too uniform. That is, all nodes within a given branch share the same direction vector. If we study the branch of a real tree, we know it is far from perfectly straight. Usually, it bends multiple times along its length. As we have already added the functionality to deviate a branch node's direction vector, we can widen the application of this technique to deviate every node in the tree from its parent's direction vector.

We will want to control the deviation such that, the deviation applied to a normal branch node's vector is not as strong as the one applied to a node that is the start of an entirely new branch. After all, if every single node in every single branch could deviate by a large amount, we might end up with branches that look more like springs. We can control this with two sets of deviation input parameters to the system. The first pair of input parameters describes the deviation cone that can occur between segments within the same branch. The second pair of parameters describes the deviation cone that can occur for the root node (the starting segment) of any branch. Typically, we will want the deviation cone used for segment deviation within the same branch to be quite small, as shown in Figure 12.11.

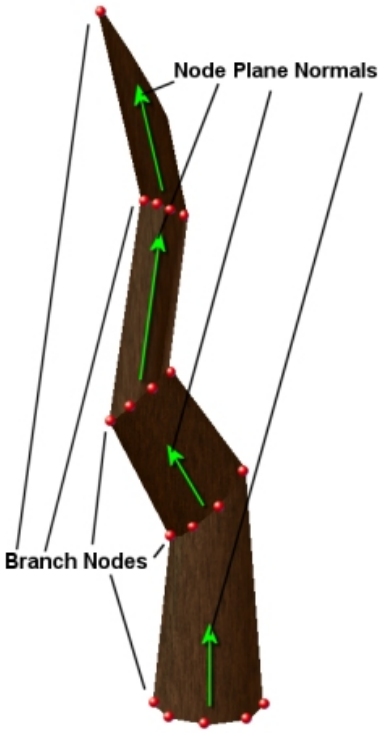

Figure 12.11

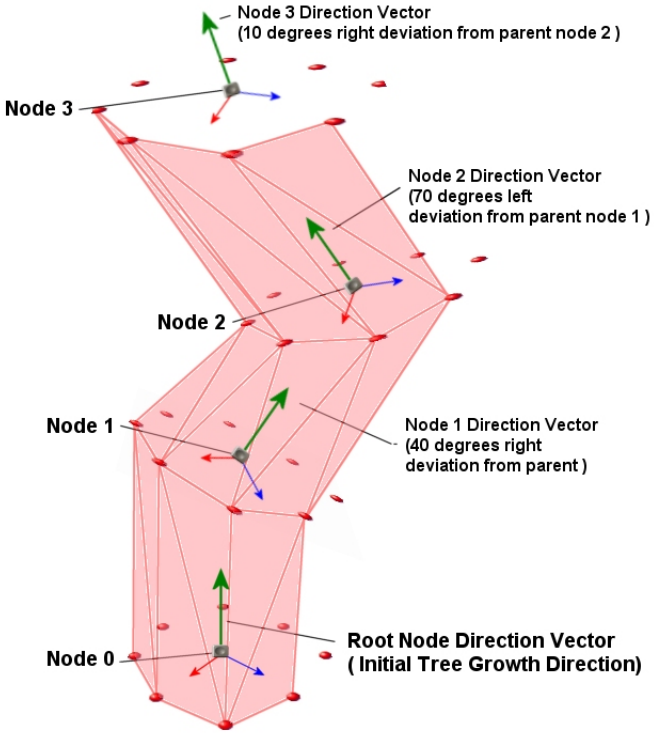

Figure 12.12

Note: Figure 12.11 and 12.12 show two different branches. They are not intended to depict the same vector deviations at each segment.

Figure 12.11 shows how a single branch mesh would be constructed if per-node direction vector deviation was used during the creation of the virtual tree hierarchy. In this diagram, you can see that the initial node had a direction vector (0,1,0) and thus a ring of vertices was inserted and aligned to the XZ plane. However, when a new node is generated to extend the branch, its direction vector is calculated by deviating the parent node by some random amount (within a range specified by input parameters). In this example, you can see that the second node deviated the parent node direction about 35 degrees to the left. Notice that when we come to insert the vertices for this node, the direction vector (along with the two tangent vectors), describes the plane on which the vertices must be positioned. Once again, we can imagine that the node position describes the initial point on the plane at which we add vertices, before then translating them out along the plane (using the tangent vectors) into their positions in the ellipse shape (see Figure 12.12). Looking at the third node (third row of vertices) in Figure 12.11, we can see that this node was deviated from its parent about 45 degrees to the right. This process is repeated for every node of every branch.

So we have learned that when we build our node hierarchy, we will start with the initial growth direction vector of the root node. This vector will be passed through a recursive procedure and deviated for the next node, which will pass its new direction vector onto its child node where it will be further deviated, and so on. Each iteration of the node hierarchy building process will involve, in simple terms, positioning a new child node, and calculating a direction vector for that node by deviating the parent vector.

When we combine the lesser per-segment deviation with the more extreme deviation performed to calculate the direction vector for each starting node in a new branch, we end up with a virtual tree representation that. when converted to mesh form, is a lot more pleasing (see Figure 12.13).

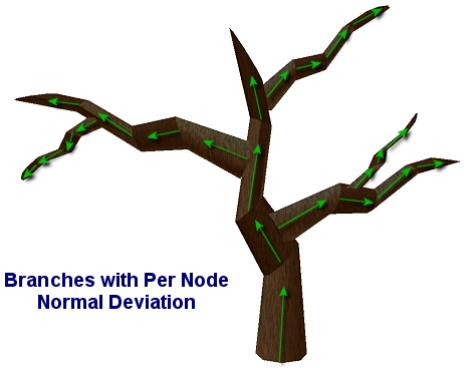

Figure 12.13

Figure 12.14 depicts how the node hierarchy for this tree might look after it has been created. For reasons of image clarity, we have slightly offset the nodes that start new branches from the position of their sibling nodes. In reality however, they would occupy the same position.

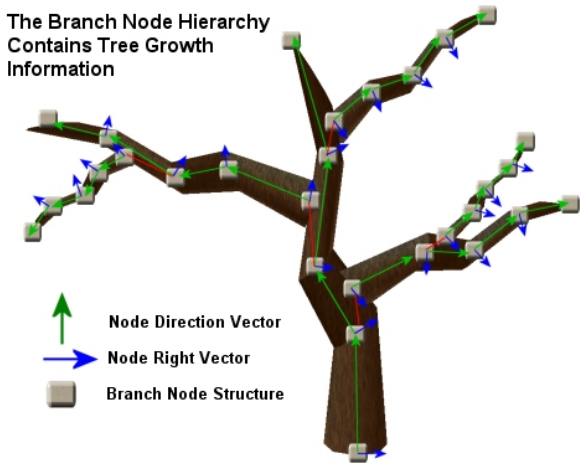

Figure 12.14

If you examine Figure 12.14 you will notice that each node contains a direction vector depicted by the green arrows, and a right vector that is perpendicular to the direction vector. We will also pass the right vector of the initial segment into the system in addition to the direction vector. Why do we need to pass this right vector through the tree? We need it in order to deviate the direction vector of each node during the recursive process. This will be explained in the next section.

12.2 Direction Vector Deviation

Before we examine the source code, we have one major process left to discuss on a theoretical level. Given a vector A, how do we generate a new random vector B that is contained within a specified cone set up around the original vector? It is easy in the two dimensional case since we merely have to rotate the new vector either left or right of the original vector by a random number of degrees. However, when working in three dimensions, simply rotating the vector left or right, or backwards or forwards will not allow us to generate a new vector that is anywhere within the specified cone.

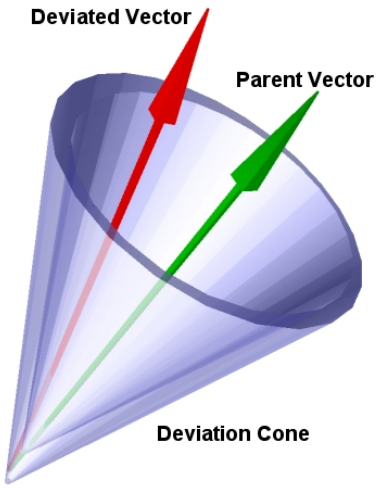

Figure 12.15 : We need to generate a random vector anywhere within the deviation cone.

Figure 12.15 demonstrates the problem we need to solve. The green arrow running down the center of the cone is assumed to be the parent vector (i.e., the vector we wish to deviate). The cone itself represents the range over which the vector can deviate. The goal is to generate a totally random vector anywhere within this deviation cone (the red arrow is one example of a possible solution). As can be seen, simply rotating the original vector clockwise or counter clockwise about a single axis will not provide us with a means to accomplish our objective.

You will see in a moment, when we discuss the structures that we are going to use in our code, how the size of this cone is specified using an angle. However, we will also have another variable that influences vector deviation called *deviation rotation*. Describing what this variable is will allow us to also understand the question at hand, "How do I generate a random vector with a cone?" The answer to this question will explain why we need to calculate a right vector at the root node and pass it through the recursive process. Each node will have its own right vector which will be used in the deviation process of its child nodes.

We will use a simple example to demonstrate the process of deviating the vector. While the same deviation function is used for the deviation of normal segment nodes and for nodes starting new child branches, in this example, we will show how a new branch is spawned from a node and how that node's direction vector is deviated to create the direction vector of the new branch start node. To give clarity to these examples, we will show the mesh data that will eventually be created from the virtual node. However, just remember that the node hierarchy that we generate will not contain any mesh data; it will be a hierarchy of data structures containing positional and directional information that will later be used to describe a tree shape to the mesh building process.

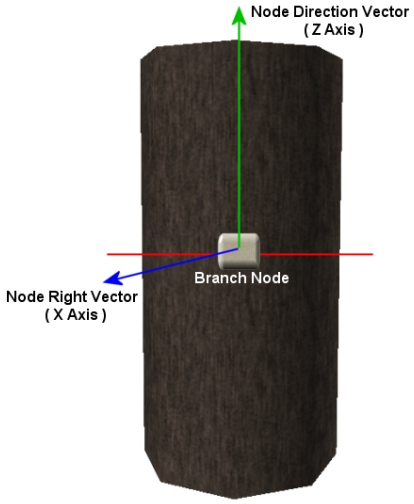

Figure 12.16

In Figure 12.16 we see a single node that is situated at some arbitrary position along a branch. We will assume this is a node in the root branch. Notice how the node stores a direction vector (the local Z axis of the node) and a right vector (the node's local X axis).

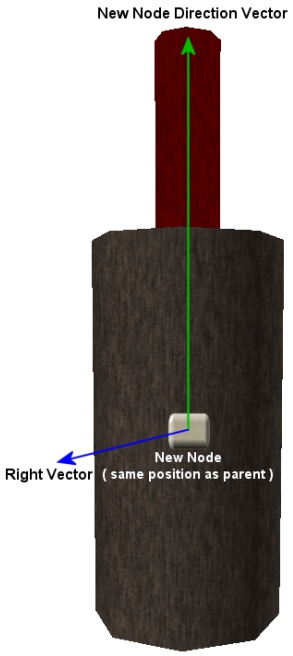

Figure 12.17

Next, let us assume that while processing this node we determine that we would like to create a new branch at this node. In Figure 12.17, the new node is added at the position of the parent node and currently has inherited its direction vector. In the diagram the red cylinder mesh sticking out of the top of the parent segment would not actually exist at this point, but will make our explanation easier. We can think of this for now as being the branch segment that will eventually be built from the branch start node we have just created.

In this diagram, the right vector of the parent node is assumed to be coming out of the page. You will notice that this provides an axis around which we can rotate the new branch node's direction vector. For example, if the cone deviation angle was 90 degrees, then we could generate a matrix that will rotate the new node's direction vector about the parent node's right vector by some angle between 0 and 90 degrees (to allow randomness). In fact, that still is not quite what we want since that would only rotate the vector one way (right for example). What we want to do instead is choose a random number between –cone angle/2 and +cone angle/2. For example, if our input parameters describe to the system that a new child branch node should be deviated by 90 degrees, we would generate a random number between –45 and +45 and use this value to build an axis rotation matrix around the parent node's right vector. This will allow for counter-clockwise and clockwise rotations within the cone.

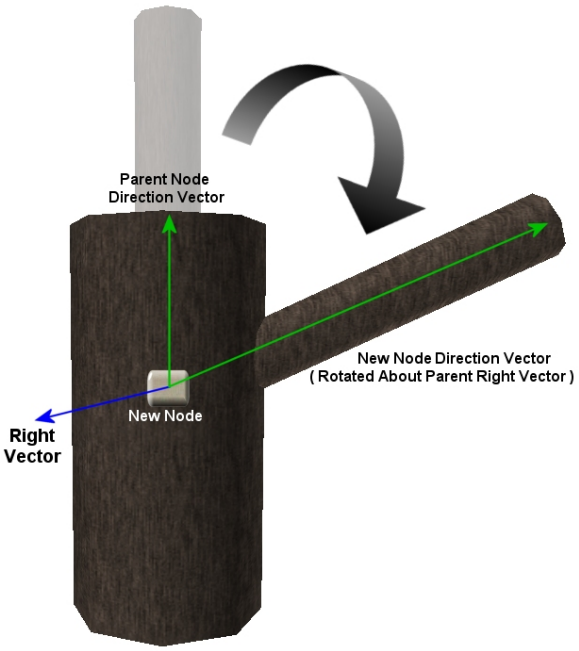

Figure 12.18

After rotating the child node's direction vector about the parent node's right vector (Figure 11.18), we will perform a second random rotation of that vector about the parent node's direction vector. This vector acts as a local up vector for the child node allowing us to rotate its newly rotated direction vector to any position around the circumference of the parent branch (see Figure 12.19). This means, our vector deviation routine will require an additional angle range that describes the permissible rotation that can be applied to the child node's direction vector when generating its random position on the branch. Usually, we will want to give branches an equal chance of shooting out from the parent at any angle, and as such, would set the up axis rotation range to 180 degrees. Then, when we need to rotate the child direction vector about the parent direction vector, we simply choose a random angle between 0-180 to rotate the new node into its final position.

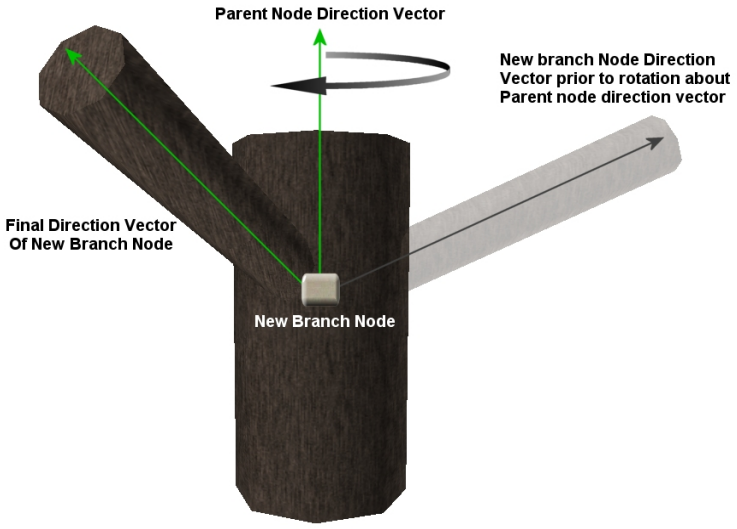

Figure 12.19 : Polar Rotation

Why do we only need to rotate 180 degrees in the second rotation step instead of 360? Remember that in the first step we will rotate the branch either counter-clockwise or clockwise depending on whether a positive or negative angle is used for rotation about the right vector of the parent node. Therefore, this first step essentially rotates the child node's vector into either the negative or positive half of a circle surrounding the parent node. Therefore, the first step chooses the semi-circle which the branch will grow from, and the second step allows us to access any random angle within that semi-circle. Of course, you can limit the rotation angle if you want branches limited to a smaller range of values on either side of the circle. Usually though, in the case of deviating the vector of a new branch start node, you will want to allow full 180 degree rotation on either side so that the branch can grow from anywhere. It should be noted that the same deviation procedure is used for deviating both the direction vectors of new branch start nodes and for the deviation between nodes within the same branch. However, as we will usually want segment to segment deviation within the same branch to more closely follow the overall growth direction of the branch, we will generally use much smaller deviation angles for both rotation steps in the process. With this two step deviation approach we essentially describe a virtual cone with the parent node's direction vector at the center. The new deviated vector will exist somewhere within this cone (Figure 12.20).

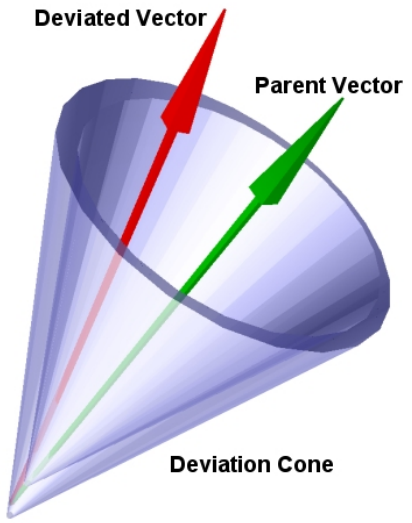

Figure 12.20

In the data structure that we feed into our CTreeActor::SetGrowthProperties function (prior to calling the CTreeActor::GenerateTree function), we will specify two sets of ranges. We will set an axis rotation range for both up and right vectors for node to node deviation within the same branch, and we will also set an axis rotation range for both of the vectors that the deviation process will use when calculating the direction vector of the node that is to become the first node in a new branch. By separating these into two separate deviation cases, we can easily control new branch and inter-branch deviation.

Looking at Figures 12.18 and 12.19 we can see that having the right vector stored at every node is important during the initial growth of the virtual tree as it is needed to deviate the directions of any child nodes. Therefore, while we have shown that the direction vector of each node must be deviated from the parent to create a new direction vector for the child node, once this has been accomplished, we must also calculate the right vector of the new node.

As it happens, updating the right vector in the deviated node is simple since we have already generated the matrix to rotate it into its new position. When we performed rotation on the child nodes direction vector, we would have rotated about the parent node's direction vector by some random amount. We will use 90 degrees in this example. So all we have to do is rotate the parent node's right vector around the parent node's direction vector and we have the new right vector for the child node. The basic node building process now looks like this:

We start the process by passing in a single growth direction vector for the tree (e.g., <0,1,0>). We will perform the cross product with this vector and the world axis that is least aligned to it to create the right vector. We now have the growth direction vector and the right vector for the first node in the tree. From this point on the recursive process starts and works as follows:

1) Will this node generate another child node in the same branch (another branch segment)?
    a) Generate a new segment node and attach it to the parent as a child
    b) Deviate the parent node's direction vector and right vector using (small random values)
    c) Store deviated vectors in the child node

2) Will this node generate a new branch?
    a) Generate a new branch node and attach it to the child list
    b) Deviate the new node's direction vector and right vector using (large random values)
    c) Store deviated vectors in new child node
    d) Repeat steps a-c for each new branch generated at this node
3) Repeat steps 1 and 2 for newly generated node(s)

When we set the deviation properties for our system, we will in fact have three range values we can specify for each of the two deviation types (segment deviation and new branch deviation). We will have a minimum cone angle, a maximum cone angle and the polar rotation angle. The minimum and maximum cone angles (which are a pair) are used to influence the first rotation step (the rotation around the parent's right vector). These values instruct the deviation function to generate a random deviation angle for the first rotation step that is no greater than the maximum cone angle, but also, no smaller than the minimum cone angle. This allows us to set a minimum level of deviation to ensure that we always get at least some deviation. When setting the minimum and maximum cone angle for the new branch deviation settings, the variables can greatly effect the shape of the tree. For example, by using a large minimum angle and only a slightly larger maximum angle, you can generate a tree that has a very uniform looking branch growth direction (like a conifer tree).

By introducing a minimum cone angle variable combined with the polar rotation step, we essentially specify a region inside the cone where vectors can be produced. It is perhaps more accurate to say that we define a region at the center of the cone where random vectors *cannot* be produced (Figure 12.21).

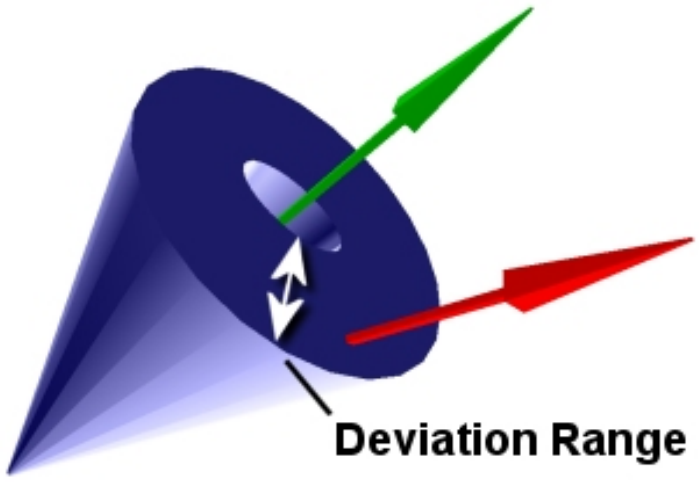

Figure 12.21

While this might sound like a complicated process, you will see shortly that the CTreeActor::DeviateNode method is only a few lines of code long.

12.3 Building the Virtual Tree

We now have all the theoretical knowledge at our disposal to discuss the code to the first part of the tree generation process. This will be the process that essentially grows the virtual tree one node at a time. Remember, at this point we are not dealing with mesh or bone data at all; we are simply building a hierarchy of data structures that contain direction vectors, right vectors, and dimensions.

Let us first see how the application might generate a tree using our new CTreeActor class.

```
// Create a new CTreeActor
CTreeActor * pTree = new CTreeActor;

// Fill out a TreeGrowthProperties Structure
TreeGrowthProperties tgp;

// Fill in the TreeGrowthProperties structure here
// NOT SHOWN HERE AS WILL DISUCSS THESE PROPERTIES IN A MOMENT

// Send out properties to the actor
pTree->SetGrowthProperties ( tgp );

// Build everything
D3DXVECTOR3 RootSegSize( 2.0 ,2.0 ,2.6 );
pTree->GenerateTree ( RootSegSize )
```

That is all there is to it. We create an instance of CTreeActor and then we fill in a TreeGrowthProperties structure. We will look at all the members in this structure in a moment. For now just know that it influences the way the nodes of the virtual tree are grown. We then send this structure to our CTreeActor so that it will have access to these properties during the tree generation process. Finally, we call the one function that makes it all happen: GenerateTree. This function first builds the virtual tree of information nodes (which we call *branch nodes*). Note that we pass in a vector of dimensions that describe the three radii of an ellipsoid that will contain the first branch segment (the root node). The X and Y components describe a box that will bound the ring of vertices generated for that node. If X and Y are equal, the branches will be perfectly circular. However, if we assign X and Y different values such as 2 and 5 for example, the thickness of the root segment of the root branch would be two tree space units in the X dimension and 5 tree space units in the Y dimension. As tree space is essentially the world space coordinate system, but with the tree at the origin, these dimensions directly describe the thickness of the root segment in world space (provided you do not transform the actor into world space using a scaling matrix). The Z component of the dimension vector describes the length of the root segment of the root branch. That is, when building the root branch (the trunk), this is the distance along the root node's direction vector that the second node will be positioned. This equates to the location of the second ring of vertices during the mesh building process. You will see when we examine the recursive tree building process, that the dimensions of each branch node are scaled down versions of the dimensions inherited from their parent. This ensures that branches get thinner and shorter as they near their end.

When the virtual tree has been completely built, we will then enter the second phase of tree construction. We will traverse the virtual tree hierarchy, creating vertices at each node and adding them to the meshes for the branch to which that node belongs. While building the meshes for each branch, we also assemble (using the node hierarchy) a D3DXFRAME hierarchy that will be used as the bone system for the tree.

Finally, when the actor's hierarchy and meshes have been built, we create the actor's animation controller and add optional animations to simulate the tree branches swaying in the wind. Once the GenerateTree function returns, the tree is complete and we can use it in our application just like a regular actor, or we can save it out to an X file for import it a world editor. Using the CTreeActor class within your application (should you wish not to save the generated tree out to an X file and load it in as a regular actor) is no different from using a normal actor. That is, you call its AdvanceTime method to update its animations each frame update and use its DrawSubset methods to render it. The virtual tree hierarchy is no longer needed at this time since it was only used to generate the frame hierarchy of bones and the mesh data for each branch.

12.3.1 The TreeGrowthProperties Structure

So that we get a better feel for the system, the best place to start is the TreeGrowthProperties structure since it defines the behavior of the CTreeActor during the virtual tree building process. This structure is pretty large as the system has many variables that you can tweak to provide you with a means to generate a vast number of different tree configurations. Not all of the members of this structure will make immediate sense until we see them being used. So a detailed discussion of some of them may be deferred until the actual mesh building process is covered.

```
typedef struct _TreeGrowthProperties
{
    USHORT      Max_Iteration_Count;
    USHORT      Initial_Branch_Count;
    USHORT      Min_Split_Iteration;
    USHORT      Max_Split_Iteration;
    float       Min_Split_Size;
    float       Max_Split_Size;

    float       Two_Split_Chance;
    float       Three_Split_Chance;
    float       Four_Split_Chance;
    float       Split_End_Chance;

    float       Segment_Deviation_Chance;
    float       Segment_Deviation_Min_Cone;
    float       Segment_Deviation_Max_Cone;
    float       Segment_Deviation_Rotate;

    float       Length_Falloff_Scale;

    float       Split_Deviation_Min_Cone;
    float       Split_Deviation_Max_Cone;
    float       Split_Deviation_Rotate;

    float       SegDev_Parent_Weight;
    float       SegDev_GrowthDir_Weight;
```

```
    USHORT      Branch_Resolution;
    USHORT      Bone_Resolution;

    float       Texture_Scale_U;
    float       Texture_Scale_V;

    D3DXVECTOR3 Growth_Dir;
} TreeGrowthProperties;
```

This structure may seem a little daunting at first, so we will examine the members one at a time. You will see that many of them relate to the concepts we have already discussed. They act as a means for setting ranges in which certain behaviors can and cannot happen during the growth process.

USHORT Max_Iteration_Count

The maximum iteration member allows us to control the overall depth of the virtual tree hierarchy we create. We start at the root node with an iteration count of 1. Every time a new segment/node is generated for a branch, the iteration count is increased and assigned to the child node. This recursive process continues such that, with every node within a given branch, the segment count is increased. As soon as (or if) the iteration count of the current node reaches the Max_Iteration_Count, no new segments/nodes will be generated along the branch and the branch will be capped and ended. One thing to watch out for is that when new child branches are spawned, the first node in that branch inherits the same segment count as the sibling node of the parent branch. Remember, if a node spawns three child nodes, there will be a sibling list of four nodes in the hierarchy -- three branch start nodes and the node that spawned the branches (the next segment of the current branch being processed). All nodes within the sibling list will contain the same iteration number (see Figure 11.22).

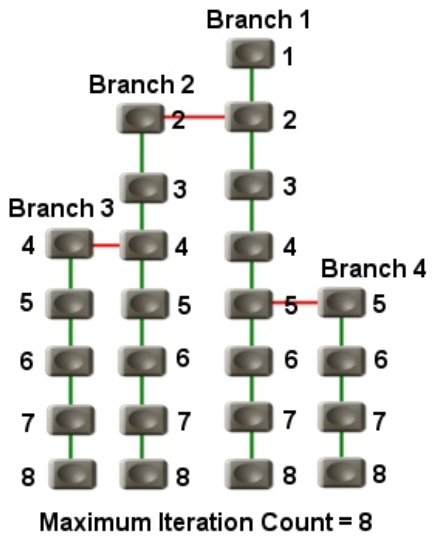

Figure 12.22

Figure 12.22 shows an example hierarchy for a tree that is constructed from four branches. The first branch is, of course, the trunk; but notice that when we add the second node of the trunk we also decide to start a new child branch. The second node of the trunk and the first node of the child branch are in a sibling list, both with an iteration count of 2. This iteration count is then continued down each branch. As you can see, the fourth branch starts with an iteration count of 5 and terminates at 8, being only four segments in length. In this example, we use a maximum iteration count of 8, so the virtual tree hierarchy will never be deeper to traverse than eight levels.

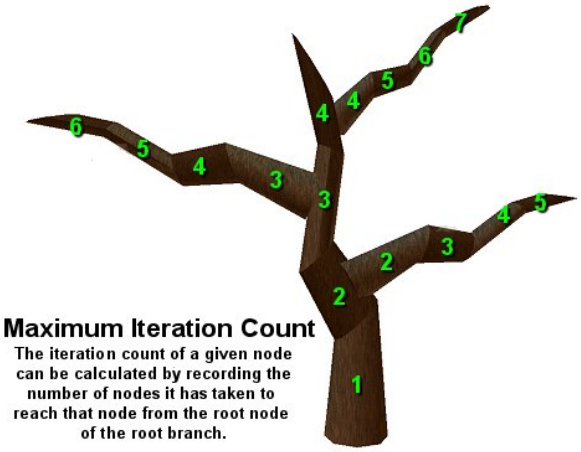

Figure 12.23

Figure 12.23 shows the tree in mesh form so that we can see how the iteration counts apply to the various segments. We are using a bit of artistic license here since we are placing the node iteration count values in the branch segments. We know that in reality however, the nodes that will contain those iterative values will be positioned at the branch segment boundaries (two of which create a branch segment). Nevertheless, this should provide an easy way to see how the iteration count can be used to set a maximum tree depth on the recursive procedure during the generation of the virtual tree.

It should be noted, that while Figure 12.22 clearly shows each branch terminating at the maximum iteration count threshold, this will not always be the case. There are many factors that are considered when deciding to add another segment/node to a branch or whether to terminate the branch at the current node. Such factors are the current branch thickness and of course a random element thrown in for good measure. Therefore, if we set the maximum iteration count to 20, we will not generate a tree where all branches end at the 20^{th} level in the hierarchy. Some branches may be terminated much sooner due to other factors. However, it is one of the many limiting factors that we will place on the procedure to control the depth of our hierarchy and the ultimate the size and complexity of our tree. There will never be a single path of branch segments from the root segment in the root branch to any branch's terminating segment that will cross segment boundaries more times than the value we set here.

USHORT Initial_Branch_Count

In all the example trees we have examined so far, we have assumed that the tree had a single initial branch (the trunk) from which all others were spawned. However, there is no reason why this has to be a strictly followed rule. While it is clear that your trees will require one initial branch (the trunk), you may wish to model trees that have multiple trunks (Figure 12.24). This parameter allows us to specify the number of initial branches we would like to tree to have.

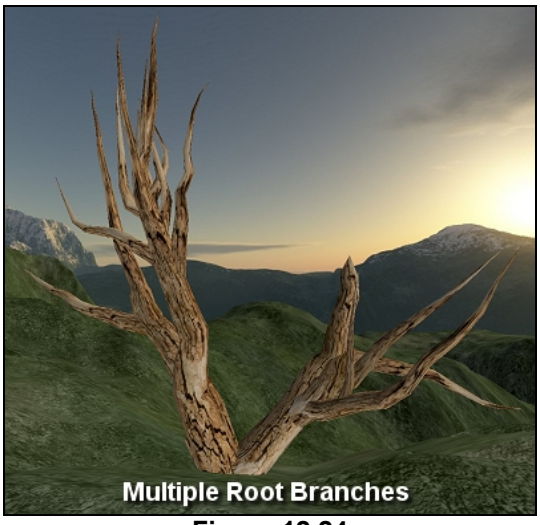

Figure 12.24

When multiple initial branches are enabled, all initial branches start at the same position (0,0,0) in tree space. Normally, when a single initial branch is being used, the direction vector we pass into the CTreeActor::GenerateTree function will be used 'as is' for the direction of the first node in that branch. When multiple initial branches are being used, each initial node in each trunk branch we create will be randomly deviated from the initial vector passed in. For example, we might imagine when looking at Figure 12.24, that when GenerateTree was called, an initial growth direction vector of <0,1,0> was specified. However, we cannot assign this same initial vector to the first node in each root branch or they would be created in exactly the same position and facing the same direction (thus, we would only see one branch). So instead, the passed vector is randomly deviated to create the actual initial direction vectors of each branch root node. In Figure 12.24 you can see that the node direction vector for both root branches has been rotated left or right to some degree from the initial direction vector passed in.

From the perspective of our virtual tree hierarchy, it simply means that there will no longer be only a single root node in the first level of the hierarchy (describing the start node of the only trunk). Instead, the first level of the hierarchy may consist of a sibling list of nodes, where each node in the list describes the start node of a trunk branch. By default, the Initial_Branch_Count member is set to 1 (in the constructor). By setting it to higher values, we are able to model foliage that grows in a more clustered manner (e.g., a rhubarb plant).

> **Note:** CTreeActor can be used to model foliage types beyond trees (bushes, shrubs, etc.) using the same methodology. If you did not want to include the overhead of all of the hierarchy and animation and skinning for small shrubs, you can still use CTreeActor to generate the shrub and then save it as an X file. When you load it back in, you can load it as a standard mesh (CTriMesh) rather than as an actor. We will talk more about this later.

USHORT Min_Split_Iteration
USHORT Max_Split_Iteration

These two members provide us with a means to control the range in which child branches are allowed to be spawned. For example, we will often *not* want a child branch to be spawned at the first segment of the trunk (during the first iteration of the recursive process).

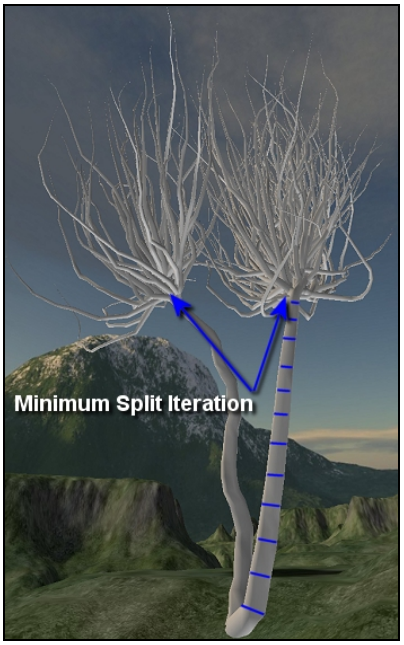

Figure 12.25

As discussed above (see Max_Iteration_Count), the iteration value is increased as each child node is added to the virtual tree hierarchy and it is directly related to the current branch segment that will be added to the mesh. We can think of the iteration count of a given node as indirectly describing the level in the hierarchy at which it resides. Figure 12.25 shows an example where we have set the minimum split iteration count quite near to the maximum iteration count of the tree such that splits into child branches only occur at the very top of the tree. Simply put, when a new node is added to the virtual tree hierarchy, we will decide whether or not we should split at this node and create multiple child branches. Splits will never happen if the iteration count of the current node being processed is smaller than Min_Split_Iteration or larger Max_Split_Iteration. Therefore, a node will only be considered for splitting if its iteration count (its depth in the hierarchy) is within the range described by these two members.

float Min_Split_Size
float Max_Split_Size

These two values allow us to further define a range of branch sizes with which child branches can be spawned from a node. When we call CTreeActor::GenerateTree, we will pass in a vector describing the size of the initial root node. As previously discussed, the X and Y components of this vector will describe the radii of the circle/ellipse that will be used to position the vertices at that node when we build the mesh. With each iteration, when we add new child node, we will subtract a small amount from this vector so that the vertices placed at each node define a smaller ellipse the further along the branch we move. This will produce a tapering effect. At the very end of a branch, the ellipse becomes a single point defining the tip of the branch. Essentially, by dividing the dimensions vector of the root branch node by the maximum number of iterations of the tree, we have a value that we can subtract from the dimensions of a parent node to generate the dimensions of the new child segment. By using this value, we can be sure that any branch will get progressively smaller as its length is traversed, with the final end node in a branch being the tip.

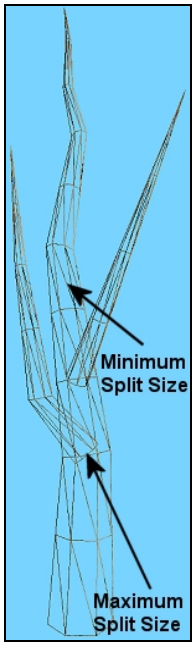

Figure 12.26

Near the end of a branch, its segments will typically be very thin, and as such, generating new child branches from these would result in extremely slim branches. They would be so thin that it would be a waste of time rendering them, especially when we consider that we will eventually add leaves to this tree that will completely obscure such microscopic detail. These two values allow us to define a thickness

range. If a node's dimensions are within this range, then it is possible for it to split into a child branch at this node. However, if the node thickness is smaller than the minimum split size or larger than the maximum split size, splitting the node will not be considered.

Figure 12.26 shows an example of a very simple tree where the Min_Split_Size and Max_Split_Size properties have been set to a very limited range in the center of the tree. As you can see, the Max_Split_Size variable allows us to control child branches not being spawned too close to the base of a branch, while the Min_Split_Size allows us to cease splitting the branch once its thickness has decreased past a sensible level.

These two members and the previous two members can be used together to provide a very flexible control mechanism for determining when and where child branches are spawned (and thus, control the shape and complexity of the tree).

float Two_Split_Chance
float Three_Split_Chance
float Four_Split_Chance

When building our virtual tree hierarchy, at every node we will need to decide whether or not we wish to introduce a new child branch. Of course, if the current iteration (tree depth) is such that it is outside the range described by the Max_Split_Iteration and Min_Split_Iteration members, then the node will not be considered for splitting. Furthermore, if the thickness of the node (its X and Y dimensions) is outside the range described in the Min_Split_Size and Max_Split_Size members, then once again, the node will not be considered for splitting. If however, the node is within the specified ranges for a possible split to occur, then we will use a random procedure to decide whether a split will occur at this node. In fact, we will perform three tests to decide whether the node should spawn two, three, or four branches (or just the one default).

These three properties each contain a percentage score that describe the probability of each node splitting into two, three, or four branches respectively. The tests are actually performed in order; we first perform a test to see if two branches should be spawned, then we perform a test to see if three branches should be spawned and finally we test to see if four branches should be spawned. The following pseudo code should give you the basic idea until we get the actual code.

```
// Generate Random Number Between 0 and 100
if ( RandomNumber < Two_Split_Chance)    NewNodeCount=2;
if ( RandomNumber < Three_Split_Chance)  NewNodeCount=3;
if ( RandomNumber < Four_Split_Chance)   NewNodeCount=4;
```

A random number will be generated between 0 and 100. Our three split chance members should be defined in that range too. Note above that all tests are always performed, so we always fall back to the highest number of splits should more than one of the tests succeed. For example, assume that we set our Four_Split_Chance member to 40 (40% chance) and our Two_Split_Chance to 80 (80% chance) and that at a given node the random number generated is 10. 10 is smaller than 80 so we have passed the two split chance and NewNodeCount is set to 2. However, when we perform the last test we also find that 10 is smaller than 40 so the Four_Split_Chance case wins out and we introduce four new branch nodes (three new branch segments plus the next branch segment of the current branch we currently processing). A different random number is generated for each test, so this is not as redundant as it

sounds. The reason we have ordered the test such that the highest number of splits takes precedence if multiple tests pass, is because typically we will set the probabilities of the four split case much lower than the two split case. Therefore, the two split case will often succeed when no others do. In the rare cases where the four split test does succeed, we want it to override the results of any previous tests.

By setting these values to different percentage values, we can influence how many branches are generated at each node. If all the tests fail, then no split will happen at this node and just the one node (the continuation of the branch) will be added.

float Split_End_Chance
This member is another percentage [0, 100] that describes the probability that each node has of ending the branch to which it belongs if it spawns one or more child branches. When it is determined that a branch node we are adding to the current branch will also spawn new child branches, we will generate a random number between 0 and 100 to determine whether the current branch we are adding should be terminated when the new branches are spawned. If the random number we generate is smaller than the Split_End_Chance probability, the current node will terminate the current branch, allowing the new child branches it spawned to continue. This value is not considered when a branch node is added that does not spawn new child branches. It really is just used to determine what the chances are of a branch terminating at the point where it forks into multiple child branches. We can think of it as the probability that the branch has been pruned at that node.

float Segment_Deviation_Chance
float Segment_Deviation_Min_Cone
float Segment_Deviation_Max_Cone
float Segment_Deviation_Rotate
These members control the range of random vector deviation that is applied to a child node's direction vector. As previously discussed, when a new node is added to continue a branch (not a branch start node), we will typically want to randomly deviate the direction of the new node so that all the segments in the branch do not end up sharing a perfectly uniform direction. The Segment_Deviation_Chance member should be set between 0 and 100 and will be used to determine the probability that a given node will deviate from its parent node's direction. If you set this to 100 for example, then a child node's direction vector will always be deviated from its parent node's direction vector making the branches of the tree look much more organic. Setting this to a value of 0 will generate a tree where all segments within the same branch share the same direction and no deviation will occur between segments within that branch.

Once it is determine that a child node's vector will be deviated, we will deviate it in two steps as discussed earlier in this lesson. First, we will rotate the child node's direction vector (initially inherited from the parent node) around the parent node's right vector. The amount we rotate this vector is described in degrees by the Segment_Deviation_Min_Cone and Segment_Deviation_Max_Cone properties. For example, if we set the minimum cone angle to 60 and the maximum cone angle to 110, then a random deviation angle will been chosen between 60 and 110 degrees. However, we do not always want the rotation to happen in the same direction around the parent node's right vector, otherwise our trees will have a tendency to all lean one way. So we convert the random angle into either a negative or positive number based on a random decision. This would allow us in this example to create a rotation between either –60 and –90 degrees or +60 to +90 degrees. Below we see the code that would

generate the rotation angle used in the first step. It generates a rotation angle between the minimum cone angle and the maximum cone angle in either the clockwise or counter-clockwise direction.

```
float fAzimuth = (float)rand() / (float)RAND_MAX;

fAzimuth = Segment_Deviation_Min_Cone +
           ((Segment_Deviation_Max_Cone - Segment_Deviation_Min_Cone) * fAzimuth);

if ( RandomNumber > 50% ) fAzimuth = -fAzimuth;
```

The first line generates a random number between 0.0 and 1.0. The second line essentially uses this value to scale the delta value between the minimum cone angle and maximum cone angle which is then added to the minimum cone angle. At this point we have a positive angle in the correct range, so the third line generates a random number between 0 and 100 and basically flips the sign if the random number is in the second half of its range. This ensures we have a 50/50 chance that each deviation will be either positive or negative in direction.

Let us plug in some values to see how that works. Let us imagine that we have set the minimum cone angle to 40 degrees and the maximum cone angle to 110 degrees. Let us also imagine that the initial random value (fAzimuth) is 0.5. This essentially means we wish to perform either a positive or negative rotation to a position halfway between 40 and 110 degrees (+/- 75 degrees).

```
float fAzimuth = 0.5; // Half way between min and max

fAzimuth = 40 +(( 110 - 40) * 0.5);
//       = 40 +(( 70) * 0.5 )
//       = 40 +( 35  )
//       = 75 degrees

if ( RandomNumber>50% ) fAzimuth = -fAzimuth;
```

That works perfectly. Once we have a value of 75 degrees in the above example, the second random number would be generated to decide whether or not to flip the sign and make this a +75 rotation or a –75 rotation.

As another example, if the random fAzimuth value we initially generated was 0.0, this means we wish to apply the minimum deviation. The minimum deviation we can perform is defined by the Segment_Deviation_Min_Cone angle. Using the same example values above, this was set to 40, which means the minimum we should deviate is 40 degrees. Let us plug it in and see if it works.

```
float fAzimuth = 0.0; // Apply minimum deviation

fAzimuth = 40 +(( 110 - 40) * 0.0);
//       = 40 +(( 70) * 0.0 )
//       = 40 +( 0  )
//       = 40 degrees

if ( RandomNumber>50% ) fAzimuth = -fAzimuth;
```

As you can see this would generate a rotation of either –40 degrees or + 40 degrees.

With the first deviation phase out of the way, our next task (having rotated the new node's direction vector left or right) is to rotate it in a circular fashion about the parent node's direction vector. This allows us to rotate the direction vector of the child so that is can protrude from the parent branch at any angle. Setting this value to 180 degrees will allow for a vector with total freedom of deviation in a 360 degree circle about the parent node.

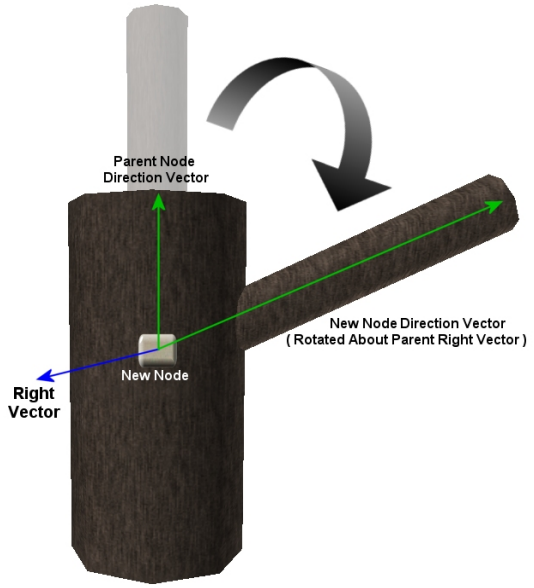

Figure 12.27
Phase 1: Random +/- rotation about parent node's right vector in the range defined by the minimum and maximum cone angles.

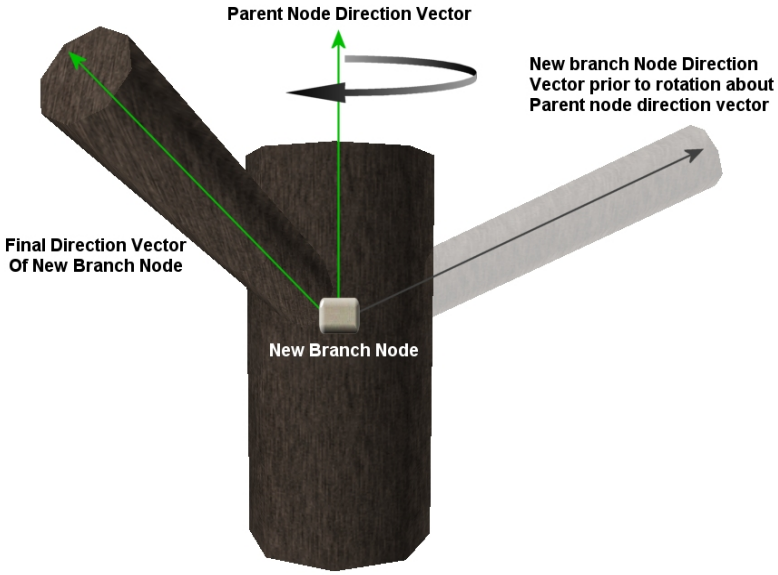

Figure 12.28
Phase 2: Random +/- rotation about the parent node's direction vector within the range: – (Polar Angle/2) to +(Polar Angle/2)

Once again, if this polar angle was set to 40 degrees, it would mean a random angle would be generated between 0 and 40. However, we do not always want to rotate about the parent node's direction vector in the same direction, so we will map it into the -/+ range. In this instance, we would want to generate a value between –20 and +20 instead of 0 to 40 using the code shown below.

```
fPolar  = (float)rand() / (float)RAND_MAX;
fPolar  = (Segment_Deviation_Rotate * fPolar) - (Segment_Deviation_Rotate / 2.0f);
```

To generate the random polar rotation angle we first generate a random number in the range of 0.0 to 1.0. We then use this value to generate an angle that is mapped into a -/+ range.

For example, let us imagine that we had set Segment_Deviation_Rotate to 30 degrees. Let us also assume that we generated an initial random number of 0.0. This should perform a negative rotation 15 degrees left as shown below.

```
fPolar  = 0.0;
fPolar  = (Segment_Deviation_Rotate * fPolar) - (Segment_Deviation_Rotate / 2.0f);
//      = ( 30 * 0.0 ) - ( 30 / 2 )
//      = 0.0 - 15
```

```
//      = -15 degrees
```

Likewise, an initial random number of 1.0 will generate a positive 15 degree rotation.

```
fPolar  = 1.0;
fPolar  = (Segment_Deviation_Rotate * fPolar) - (Segment_Deviation_Rotate / 2.0f);
//      = ( 30 * 1.0 ) - ( 30 / 2 )
//      =  30 - 15
//      =  15 degrees
```

Finally, an initial random number of 0.5 is halfway between the range and should therefore generate a rotation of 0.0 degrees (no polar rotation).

```
fPolar  = 0.5;
fPolar  = (Segment_Deviation_Rotate * fPolar) - (Segment_Deviation_Rotate / 2.0f);
//      = ( 30 * 0.5 ) - ( 30 / 2 )
//      =  15 - 15
//      =  0 degrees
```

And that is all there is to node deviation. We then use these two rotation values to build the rotation matrices with which to rotate the direction and right vectors of the child node.

float Length_Falloff_Scale

This member is one of those members that will make more sense when we look at the code. It controls how the length of each segment gets smaller as the segments near the end of the branch (the tapering effect). As discussed above, as we add each new node to our hierarchy, each node inherits the dimensions of its parent node which then has some small value subtracted from it. In the case of the X and Y dimensions of a node, these describe the radii of an ellipse on the plane described by the node, and thus the size of the ring of vertices that will be generated there. We also discussed how the value we subtract from each node is the dimensions of the root node divided by the maximum iteration count. This allows us to scale the radii of each node such that the ellipse of vertices inserted at each node will get progressively smaller towards the end of the branch (and thus the overall tree). We also do the same for the Z dimension of each node, which essentially describes the length of the cylinder segment formed by that node and its child node (the length of a branch segment). This means branch segments will not only get thinner as the tree approaches its branch tips, but also shorter.

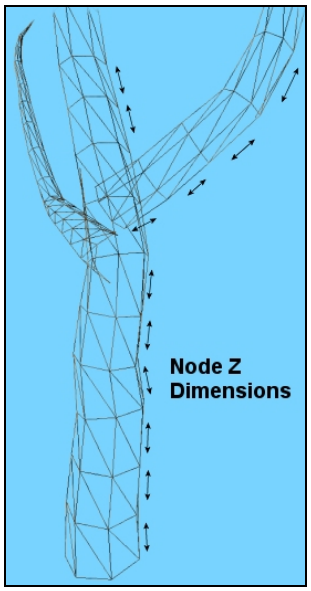

Figure 12.29

However, if you look at Figure 12.29, you can see that while we usually wish the thickness of the branch to get smaller towards its end quite quickly, usually we will not wish to scale the length of each segment at quite the same rate. In Figure 12.29 you can see that the length of each segment diminishes only slightly from segment to segment while the thickness of the each segment (the X and Y dimensions) falls off quite quickly.

The Length_Falloff_Scale allows us to control how the length of each segment (the Z dimension) gets decreased from node to node with respect to how the X and Y dimensions are scaled. For example, if we

set this value to 1.0, then the length of each segment in a branch will get smaller by the same ratio as the reduction in the thickness from segment to segment. A value of 0.5 would mean that the reduction in length from segment to segment would be half the reduction in branch thickness from segment to segment. Lower values in this member make the tree more spindly looking (long and thin) while higher values will result in tree with shorter, stumpy looking branches.

float Split_Deviation_Min_Cone
float Split_Deviation_Max_Cone
float Split_Deviation_Rotate
These values should look familiar to you. They define the cone and polar rotation used to deviate the vector of a node that is the first node in a new branch. The exact same deviation technique is used as has been previously described, and as such, these parameters are used in exactly the same way as the Segment_Deviation_Min_Cone, Segment_Deviation_Max_Cone and the Segment_Deviation_Rotate members discussed earlier. These values are used when deviating the vectors for start nodes of a new branch. This allows us to provide a much larger deviation range for new child branches which will usually sprout off from the parent at quite arbitrary angles. Contrast this with segment to segment deviation within the same branch where we usually keep vector deviation more conservative.

float SegDev_Parent_Weight
This member is used to set a weight (usually between 0.0 and 1.0) which describes how much the deviated vector of a child node should be influenced by the direction vector of its parent. Essentially, once we have deviated the vector of a node, we will add to that vector the direction vector of the parent scaled by this weight, before normalizing that vector.

```
DeviatedVector        = This is the vector that has just been randomly deviated
pNewNode->Direction   = DeviatedVector+( pParentNode->Direction*SegDev_Parent_Weight );
D3DXVec3Normalize    ( &pNewNode->Direction, &pNewNode->Direction );
```

If this value is set to zero then segment to segment vector deviation will be completely random and sporadic (within the specified ranges). By assigning this weight a value, we allow each new segment added to a branch to be randomly deviated while still following the overall direction of the branch.

It is very important to realize that the parent nodes direction vector and this weight are only used to influence the deviated vector of a child node during segment to segment deviation within the same branch. Whenever a new branch is generated, the direction vector of the first node in that branch (the root node of the branch) is not influenced by its parent node's direction vector at all. This makes sense as we often want new branches to shoot off at random angles from the parent branch. However, once the new random vector for a branch start node has been generated, all child segments/nodes of that branch will have their vectors influenced by their parent direction vectors, the first of which is the direction vector of the branch's root node. Therefore, new branch nodes are the links where parent influence is temporarily discarded (for the generation of that node only).

If you were to set this weight value to 1.0, the final vector for a new node would be the average of the deviated vector (generated in the steps discussed previously) and the parent node's direction vector.

D3DXVECTOR3 Growth_Dir
float SegDev_GrowthDir_Weight

These two members further allow us to control the overall growth direction of the tree. When we call CTreeActor::GenerateTree we pass in a vector describing the direction of the first node in that tree. From this point on, the direction vectors are passed from parent to child and deviate at each step. Therefore, even if you passed in an initial direction vector of <0, 1, 0>, this does not mean that your tree would grow in that direction at all. This only tells us that the first segment will point in that direction. As discussed in the previous parameter, we can certainly factor in the parent vector of any node into the generation of its child node's vector, however this does not always do what we want.

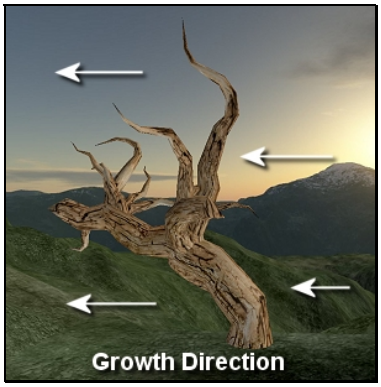

Figure 12.30

For example, if we used an initial direction vector of <0,1,0> for the root node and also set the SegDev_Parent_Weight to some non-zero value, we know that the direction vector would influence all the other segments in the root branch to some degree. This means, the root branch would generally grow in an upwards direction as described by our initial direction vector. However, what we must consider is that as soon as a new branch is spawned, the influence of the root parent branch direction vector is lost. That is, if a new branch is generated, the first node in that branch will have a totally random deviation applied to it (not influenced by its parent). From that point on, all the child nodes of that branch will be influenced by that direction vector and not the vector that we initially passed in. This is not a design flaw, since we absolutely want branches to have directions independent of their parent branches. But it would be nice if we had another level of control; a direction vector that could globally influence the direction of all nodes in the hierarchy. That is what the Growth_Dir and the SegDev_GrowthDir_Weight properties are for. These values are very useful for shaping the overall growth direction of a tree.

If you take a look at Figure 12.30 you can see a tree that has been generated with a growth direction vector of <-1,0,0> to make the tree segments generally grow in the direction of the negative X axis of tree space. In this example, a SegDev_GrowthDir_Weight of 0.2 was used. As you can see, even though we passed in an initial direction vector of <0,1,0> for the root node, as the segments are generated, the growth direction vector is still influencing the direction of every branch node. Once again, the only time that the growth direction does not influence the vector generated for a node is when the node is a root node of a branch. For example, you can see in Figure 12.30 how initially, new branches deviate in a totally random direction. As we begin to add nodes to those branches however, those child nodes are influenced by the growth direction vector and as such, the branches begin to curl round in the direction of the growth vector specified.

Let us say for example that you wanted to model a tree growing out of the side of cliff face. The growth direction for the tree could be set to <0,1,0> but the initial direction vector of the root node of the entire tree set to <1,0,0>. The tree would start growth horizontally out of the cliff face and then gradually start to grow upwards. How quickly this change in growth direction takes place depends on how you set the SegDev_GrowthDir_Weight member and the SegDev_Parent_Weight members. These members directly control, at each node, how strongly the parent node influences the node direction, and how strongly the overall growth direction of the tree does.

USHORT Branch_Resolution

This member is used to control how many vertices will be used in our node rings (and thus, our whole tree). As mentioned previously, after we have generated our virtual node tree, we will traverse that tree and insert a ring of vertices at each node. How many vertices we insert into this ring directly controls smoothness of the branches. Figure 12.31 shows the circle of vertices inserted for two nodes with a branch resolution setting of 8. As you can see, the Branch_Resolution member defines how coarse and angular the cylinder of each branch segment will be.

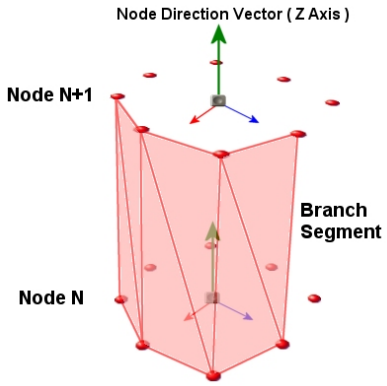

Figure 12.31

The default value is 8, which means each branch segment cylinder will be using 16 vertices -- 8 vertices at each node that form its bottom and top boundaries. Note however that vertices are shared between segments. For example, the ring of vertices inserted at node 2, form the top of cylinder 1 (Node1 ->Node2) and the bottom of cylinder 2 (Node2->Node3).

USHORT Bone_Resolution

Once we have completed the first phase of tree creation, we will have a Virtual Tree described by a hierarchy of branch node structures. This tree representation is abstracted from mesh or bone data at this point. The second phase of tree creation will involve generating the meshes for the branches themselves and building the actor's internal D3DXFRAME hierarchy which will describe the bones of the tree used to render and animate the actor.

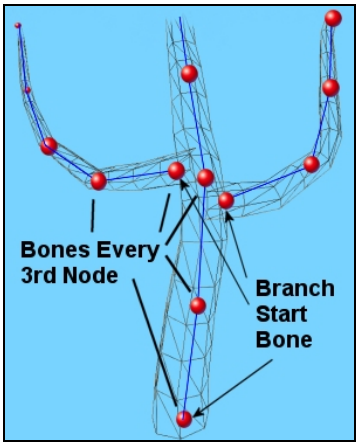

Figure 12.32

When building the meshes for the tree, we have decided that every node in the virtual tree should describe a plane that will contain a ring of vertices that both end and begin each branch segment. Therefore, each node in our virtual tree represents the actual location where vertices will be placed. We could take the same approach when building the actor's frame hierarchy, although having bones placed at every node is probably overkill for what will simply be an animation that gently moves the tree back and forth in the wind. The more bones we create for the tree, the more traversal and transformation will have to be done when rendering the tree and animating the hierarchy. Therefore, this member allows us to define the bone resolution of the tree.

The bone resolution is simply a value N that describes to the tree creation process that bones should be inserted every N nodes. The default bone resolution is 3 which means, starting from the root node, bones

will be inserted every third node that is encountered along a branch. If you look at Figure 12.32 you can see how the bones might look if placed at every third node. We have found during testing that this provides more than enough joints to move the tree in a convincing manner. In fact, the tree will still move nicely even with fewer bones than this, so you could set this member based on the end user's system to get better performance if you intended to use CTreeActor in your actual game code. As you can see by looking at Figure 12.32, the actor's frame hierarchy will contain a lot fewer frames than the node hierarchy of the virtual tree that we create. Therefore, we can think of the actor's frame hierarchy as being a discrete version of the virtual node hierarchy. This is similar to how curved surfaces are eventually turned into a discrete polygonal representation based on a continuous curve. The more polygons you allow to model the curve, the more closely the final mesh will resemble the mathematical model of that curve. While our virtual tree hierarchy is not nearly as abstract as a curved surface, it does allow us to de-couple the virtual tree generation process from the bone and mesh resolutions we ultimately end up using the build the final representation.

float Texture_Scale_U
float Texture_Scale_V

The final two members in our structure will be used to set the scale of the texture used to map the branch meshes. We learned in Module I of this series how we can assign texture coordinates values outside the [0.0, 1.0] range to allow us to repeat a texture image multiple times over the face of an object. For example, we know that if we have a texture mapped to a quad that has its texture coordinates in the 0.0-4.0 range along both its U and V axes, the texture will be mapped to the surface of the polygon sixteen times. It will be mapped four times along the U axis and four times along the V axis.

In Figure 12.33 we can see a quad that has a texture mapped to it once because its UV range is in the [0.0, 1.0] range both horizontally and vertically. In Figure 12.34, the same quad has texture coordinates in the range of [0.0, 4.0] which tiles four rows and four columns of the texture image.

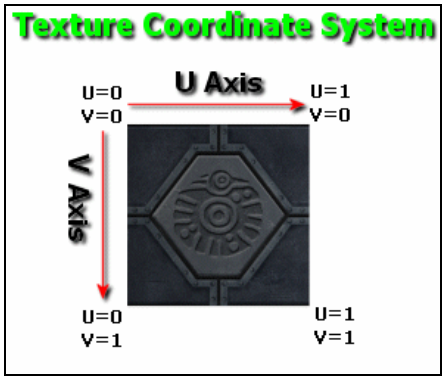

Figure 12.33 (UV range 0-1)
Texture mapped once to a quad.

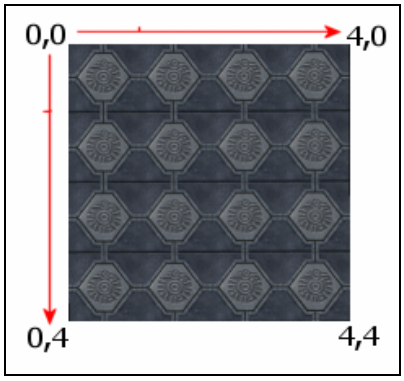

Figure 12.34 (UV range 0-4)
Texture mapped sixteen times to quad

The Texture_ScaleU and Texture_Scale_V properties of our tree allow us to apply the same scaling of the texture that we use for the branches of our tree. That is to say, the higher we set these values, the more times the texture will be tiled over the entire range of the tree. In order to understand how these properties work, we need to discuss how we will in fact generate the texture coordinates for the vertices of the tree.

Calculation of the U texture coordinate for a vertex in our branches is delightfully easy because it is a function of the branch resolution (the number of vertices we will use to form the ring at each node in our virtual tree). If the branch resolution is 8 for example, then we simply wish to assign each vertex in the ring one of 8 positions across the U axis of the texture. If we forget about the texture scaling members for the time being, we could see that the calculation of the U texture coordinate for a vertex in the ring is simply its position within that ring divided by the total number of vertices in the ring (branch resolution)

generating a value in the [0.0. 1.0] range. For example, if we have a branch resolution of 8, then we simply have to do the following calculation for each vertex in a ring:

```
for (int i=0; i< BranchResolution; i++)
{
     Vertex[i].U = i / BranchResolution-1; // zero based vertices
}
```

Let us have a look at the values this generates for each vertex in the ring.

Vertex[0].u = 0 / 7 = 0.0
Vertex[1].u = 1 / 7 = 0.142
Vertex[2].u = 2 / 7 = 0.285
Vertex[3].u = 3 / 7 = 0.428
Vertex[4].u = 4 / 7 = 0.571
Vertex[5].u = 5 / 7 = 0.714
Vertex[6].u = 6 / 7 = 0.857
Vertex[7].u = 7 / 7 = 1.0

As you can see, we have essentially unrolled the ring of vertices at that node and laid them flat across the texture. If you imagine those vertices then being pulled back into the shape of a cylinder, we have in fact wrapped the texture around the cylinder's width exactly once.

Calculating the V coordinate for each vertex is a slightly different matter but certainly no more difficult. We decided that the V coordinates should be mapped over the entire range of the tree. Essentially, vertices at the bottom of the tree will have V coordinates of 0.0 while vertices at the ends of branches will have V coordinates approaching 1.0.

We decided to do this because it allows us control over changing the color of the tree as a function of height. We could for example, have a bark texture that is very dark brown at the bottom but slowly faded to a light brown near the top. When mapped to the tree, the segment at the bottom of the tree would receive the dark brown texture colors, but as we move up the tree, the branch segments would receive the lighter portions of the texture. This might be to simulate that the branch tips are in direct sunlight and their color has been bleached. You could also generate a texture such that at the top of the texture, the bark texture has little green shoots or leaves, which once again, when mapped to the tree, the greener areas would only show up towards the branch ends.

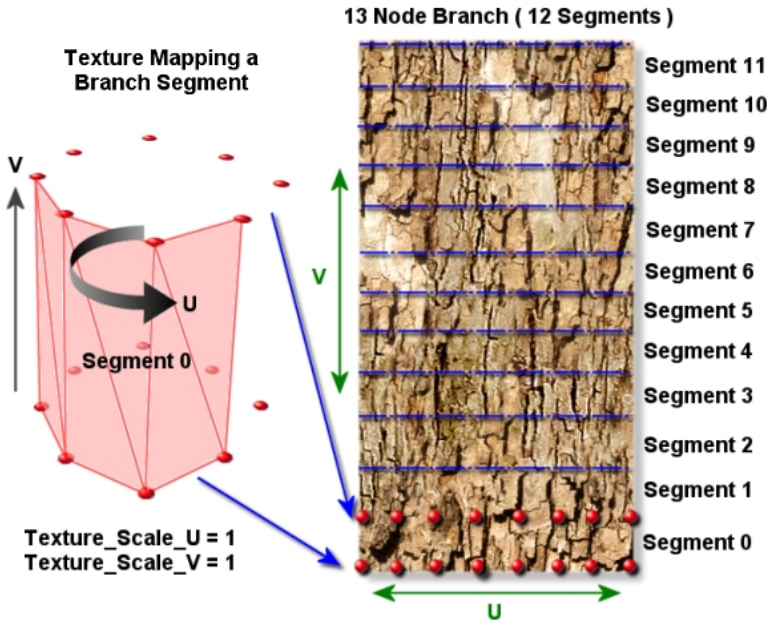

Figure 12.35
U Scale =1 : V Scale =1 : Max Iteration Count=13

In order to achieve the mapping of vertices such that the texture is mapped vertically over all segments of the tree, the generation of the V coordinate for each ring of vertices at a node, should be a function of that node's iteration count (its level in the virtual hierarchy) divided by the total hierarchy depth (maximum iteration count of the tree). Figure 12.35 shows this relationship clearly, which currently assumes a U and V texture scale of 1.0 (no scaling).

In Figure 12.35 we are shown the mapping of a texture to the first two rings of vertices (the first two nodes in the tree) which form the first segment in the root branch of the tree. For simplicity, we will also assume this tree consists of just a root branch which is constructed of 13 nodes (12 segments). If you look at the base of the bark texture, you can see how the two rings of vertices are mapped horizontally across the face of the texture using the U calculation texture we described previously. However, we can see that each node's vertices should be assigned different V coordinates which increase as the iteration of the node approaches the maximum iteration count of the tree. Using this technique, the V coordinates of the two rings of vertices forming the root segment are assigned low V values. It is also clear by looking at the segment boundaries marked on the texture that as we move up and calculate the V coordinates for the nodes deeper in the hierarchy (higher in the tree) the higher portion of the texture is mapped to the vertices of these nodes.

> **Note**: We have inverted the direction of the texture coordinate system V axis in the diagram so that it is more intuitive for this explanation. As we know, the V axis from runs top to bottom and not from bottom to top as shown here, so lower branch segments in the tree would actually be assigned the higher portions of the texture shown in this diagram, and the opposite is true.

By studying what we wish to achieve in Figure 12.35, the solution is easy. The V coordinate of any vertex is simply the iteration value of the node divided by the maximum iteration count of the tree (a property that we discussed earlier). The following snippet of pseudo code demonstrates the calculation of texture coordinates for a ring of vertices at an arbitrary node. In this example, the vertex structure is assumed to have a pointer to the branch node from which it was created. We will not implement it in this way, but this is to clearly establish the relationship between the node in the virtual tree and the vertices that are being created for it. In a moment when we look at the BranchNode structure, you will see how each node contains the iteration value that was generated during the virtual tree creation process and describes the level of that node in the virtual tree hierarchy.

```
for (i=0; i< BranchResolution; i++)
{
   Vertex[i].U = i / BranchResolution-1;
   Vertex[i].v = Vertex.pNode->Iteration / Max_Iteration_Count;
}
```

So we now know how to generate the texture coordinates for the vertices of all the branches in our tree. But what are the Texture_Scale_U and Texture_Scale_V members of the TreeGrowthProperties structure used for?

As you might imagine, they are simply used to multiply the result of the two texture coordinate calculations shown above so that we can tile the texture over the tree with a desired regularity. This upgrades our calculation code to the following final UV calculation.

```
for (i=0; i< BranchResolution; i++)
{
   Vertex[i].U = (i / BranchResolution-1) * Texture_Scale_U;
   Vertex[i].v = (Vertex.pNode->Iteration/Max_Iteration_Count) * Texture_Scale_V;
}
```

If both scaling factors are set to 1.0 then no texture scaling is done; a ring of vertices is simply mapped once horizontally across the face of the texture, and all the vertices comprising the tree are mapped once vertically along the face of the texture. By multiplying the texture coordinates by some scale value, we push them outside the 0.0 to 1.0 range, which we know will cause the texture to tile (unless texture tiling has been disabled in the API).

For example, we know that if we set the Texture_Scale_U property to 3.0, then each segment of a branch would have the texture's width wrapped around it not once, but three times. Obviously, for this to look good you must use a seamless tile-able texture. By performing scaling in this manner, the tree texture will look much higher resolution. If we were to set the Texture_Scale_V property to 2.0, then the texture would be mapped twice over the entire height of the tree. In other words, if we had a tree consisting of a single branch of 16 nodes, the entire vertical range of the texture would be mapped in its entirety to the first 8 nodes of the tree and repeated again for the second 8 nodes.

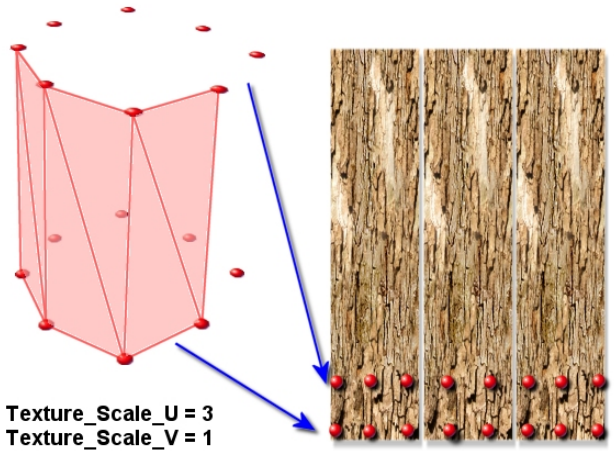

Figure 12.36

Figure 12.36 shows a simple example of tiling a bark texture with a Texture_Scale_U value of 3 and a Texture_Scale_V value of 1 (no scaling vertically). Once again, we are showing the vertex mapping for the first two rings of vertices (first segment) of the root branch. Notice how the 8 vertices now span three copies of the bark texture.

Figure 12.37 shows screenshots of the same section of the root branch of a tree mapped with this texture using different U and V texture coordinate scale values. Notice that the texture being used here is not a great texture for tiling but actually aids us in teaching this subject as it is clear that the texture is repeating across the width and height of the tree.

We can certainly see while examining the three example mappings in figure 12.37 how the U_Scale value is wrapping the texture multiple times around the section of the branch we are looking at. Of course, we are only looking at one side of the tree so we cannot see all the repeats. However, if you look at the center image, we can see a pattern starting to repeat which is much more obvious in the rightmost image. As mentioned, this texture is not ideal for tiling. But what should be clear is the extra detail that tiling seems to give the surface of our branches. The left most image, with no scaling being performed, looks quite blurry and out of scale by comparison. Of course, setting the correct texture scaling values in the TreeGrowthProperties structure will often only bear fruit by experimentation. Some textures look great when applied with no scale and some textures look awful when repeatedly tiled. Detail mapping can also be used to significantly improve the look and feel of the tree (see Module I).

Figure 12.37

We have now discussed the members of the TreeGrowthProperties structure. While that may have seemed daunting, remember that most of the last section was spent discussing how the tree will use these values in its construction. So by discussing this structure in such detail, we have also learned a great deal about how the tree will be built. This will make our job a lot easier when we examine the code. Remember, these values are set via the CTreeActor::SetGrowthProperties function, which simply assigns them to internal member variables which are accessible during tree creation. It is important that we set any tree properties before we issue the call to CTreeActor::GenerateTree since it is this function that builds the tree using the growth property information. Calling CTreeActor::SetGrowthProperties will have no effect if called after the CTreeActor::GenerateTree function.

12.3.2 The BranchNode Structure

When our application calls CTreeActor::GenerateTree method, the first task of this function is to build the virtual tree hierarchy. The nodes of this hierarchy will be arranged like D3DXFRAME structures in a frame hierarchy in that each node will contain a pointer to a linked list of children and another pointer to a linked list of sibling nodes that share the same parent. We are certainly used to this hierarchical arrangement by now. What is very important to grasp is that the hierarchy will not contain any mesh or bone data. Each node in the hierarchy is simply a packet of information, describing the position, orientation and scale of the tree at certain points. Later, this hierarchy of nodes will be used to build the actual meshes of the tree. As previously discussed, each node in this hierarchy will represent a branch segment boundary where a ring of vertices will be placed. Each node in our hierarchy will be represented by a BranchNode structure (defined in CTreeActor.h) whose members we will discuss momentarily.

It might strange that we would not build our virtual tree hierarchy out of D3DXFRAME derived structures. After all, this is the structure we are used to using and it has the child and sibling pointers that we need. It would at first seem an ideal choice to use for the nodes of this hierarchy. It is true that we will need to store much more information than a vanilla D3DXFRAME structure, but we could derive a class from this structure and add the extra members. So why do we not do this? There are several reasons we have decided to use a completely new structure for the building of our virtual tree.

The **first** reason is clarity. After our virtual tree has been constructed, we will traverse this hierarchy and build the actor's skeleton. We already know the actor's skeleton is just a D3DXFRAME hierarchy where each frame structure represents a bone. Therefore, we will be using one hierarchy (the virtual tree) to build another hierarchy (the actor's skeleton). If we implemented both of these as D3DXFRAME hierarchies, we might introduce confusion as to which hierarchy a given frame belongs to. By using two completely different hierarchy types, we eliminate the potential for bugs that could arise from accidentally connecting a D3DXFRAME structure to the wrong hierarchy.

The **second** reason is consistency. Although we can use the D3DXFRAME structure to store whatever information we choose, typically the matrix of this structure describes the position and orientation of that frame as a parent relative transformation. In order to convert that into an absolute transformation we must combine the matrices of all frames that precede it in the path down the hierarchy. We will be building our virtual tree hierarchy one node at a time using a random process and we certainly do not want to be adding nodes in parent relative space at this point. It makes much more sense to assign all the nodes a position and direction in a space shared by all nodes (tree space) so we are not constantly having to traverse the tree and perform transformations from one node's space to another. In short, we wish to store absolute positions and orientations at each node (not parent relative ones). Now it is true that we could simply use the matrix of each D3DXFRAME structure in the virtual tree hierarchy to store absolute transformations in this instance; but that is not consistent with what an application would usually expect the matrix of this structure to contain and may lead to incorrect assumptions about the matrices stored in this hierarchy if accessing frames directly.

The **third** and final reason is ease of use. It makes a little extra work for ourselves if we represent the position and direction of each node in matrix form during the building process. We have seen how we often need to work with the direction and right vectors in isolation when calculating the direction of a node and the deviation of its child nodes. Therefore, rather than having to extract the position and direction vectors from the matrix, modify them and store them back in the matrix, we will just store them in separate vector variables in the branch node structure.

The BranchNode structure also has a constructor and destructor which take care of initializing its member variables to zero and deleting its child and sibling lists, respectively. Below, we see the BranchNode structure and then discuss its member variables. Each node in our virtual tree will be a structure of this type.

```
typedef struct _BranchNode
{
    D3DXVECTOR3     Position;
    D3DXVECTOR3     Direction;
    D3DXVECTOR3     Right;
    D3DXVECTOR3     Dimensions;
```

```
    BranchNodeType  Type;
    _BranchNode    *Parent;
    _BranchNode    *Child;
    _BranchNode    *Sibling;
    USHORT          Iteration;     // The iteration at which this was generated
    USHORT          BranchSegment;
    USHORT          VertexStart;
    ULONG           UID;
    bool            BoneNode;
    LPD3DXFRAME     pBone;

    // Auto Hierarchy Destructor
    ~_BranchNode() { if ( Child ) delete Child; if ( Sibling ) delete Sibling; }

    // Auto clearing constructor
    _BranchNode()  { ZeroMemory( this, sizeof(_BranchNode) ); }

} BranchNode;
```

Let us discuss those member variables.

D3DXVECTOR3 Position
This member is where the tree space position of the node will be stored. We can think of tree space as a coordinate system where the root node/nodes of the trunk branch exist at the origin. The position, direction vector and right vectors of a node define a plane on which the ring of vertices that the node represents will be placed.

D3DXVECTOR3 Direction
This vector will contain the direction of the node. We can think of this vector as pointing in the direction of the next segment (child) in the branch. The direction vector can also be thought of as a normal to a plane on which the ring of vertices that this node represents will be placed.

D3DXVECTOR3 Right
This is where we will store the right vector of the node. This is a vector that is tangent to the plane on which the vertices will be placed and describes the direction in which the node's local X axis is pointing in tree space. That is to say, the Direction vector and the Right vector of a node represent the local coordinate system of the node (the Z and X axis respectively). By performing the cross product on these two vectors we can generate the third axis (the Y axis) of the local coordinate system for that node. This is used during mesh creation to position the ring of vertices on the node's plane.

D3DXVECTOR3 Dimensions
This vector stores the dimensions of the node. The X and Y components of the vector represent the two radii of an ellipse that has the node's position at its center. This is used to place the ring of vertices in a circle surrounding the node's position. The Z component of the vector describes the distance along the node's dimension vector to its child node (if it exists). We can think of the Z component of node N as defining the world space length of the branch segment (cylinder) formed by nodes N and N+1.

BranchNodeType Type
Each node in the tree will be one of three types of node which we must distinguish between when deviating vectors and building the mesh data. A node can either be a Branch Begin node, which means it

is the first node in a new child branch. It may alternatively be a Brand End node which means it is the terminating node in a branch and the node at which we will create only a single vertex (instead of a ring of vertices) forming the tip of the branch. Most nodes in the tree will be nodes of the third type: Branch Segment. These are nodes at some position in a branch between the branch begin and end nodes. It will become obvious when we cover the code why we must distinguish between the three node types in many places during the tree building process.

This member will be assigned one of three members of the BranchNodeType enumeration, which is defined in the CTreeActor namespace as:

```
enum BranchNodeType { BRANCH_BEGIN = 1, BRANCH_SEGMENT = 2, BRANCH_END = 3 };
```

_BranchNode *Parent
_BranchNode *Child
_BranchNode *Sibling
These three pointers are used to connect the branch node to the hierarchy. The Child and Sibling pointers mirror the functionality of the D3DXFRAME structure members of the same name. The Child pointer points to the first branch node in a linked list of child nodes in the next level of the hierarchy. If this node had a child node which had three siblings, this pointer would point to a linked list of four branch nodes. The parent node's Child pointer points to the child node at the head of the linked list, and each of the child nodes are connected by their sibling pointers. Likewise, the Sibling pointer will point to branch nodes that share the same parent node. The Parent pointer is not available in the D3DXFRAME structure and simply points to the node's parent branch node. This allows us to traverse up and down the levels of the hierarchy with ease from any given node.

As shown in Figure 12.22, the child pointer will point to a linked list of sibling nodes in the next level down in the hierarchy. This child list contains the next node in the current branch and any Branch Begin nodes of branches that start at that child node. The Sibling pointer will point to a list of nodes that exist at the same level and share the same parent. For example, let us imagine that node N in a branch has a child pointer to node N+1 which represents the next node in the branch. Also imagine, that at node N+1 two child branches are started. Node N would have a child pointer that would point to a linked list of three nodes: Node N+1 in the current branch and the two Branch Begin nodes for the new branches spawned at node N+1. These nodes would be connected via their sibling pointers.

USHORT Iteration
As discussed previously, as we step through a branch adding child branch nodes, the iteration count will be incremented and passed to the child nodes. The iteration basically describes the level in the hierarchy that the node exists at. For example, Node N would have an iteration count of N and any child nodes would have an iteration value of N+1, and so on down the tree. The Iteration of a node is also used in calculating its V texture coordinate as discussed in the previous section.

USHORT BranchSegment
This member will contain the Branch Begin relative index of the node. In other words, it stores the zero based node index from the start of the branch to which it belongs. Whenever a new branch begins, a new Branch Begin node is added with a BranchSegment value of zero. The next node of this branch will have a value of 1 assigned to its BranchSegment value, etc. We can think of this in many ways as being the branch local equivalent of the node's Iteration member. The node's iteration describes the number of

nodes that would have to be traversed from the root node of the root branch to reach the current node in the tree. The BranchSegment value describes the number of nodes that would have to be traversed from the first node (Branch_Begin node) in the current branch to reach the current node. This value will be useful when generating the mesh data for a given branch. Remember, each branch will be a separate skinned mesh.

USHORT VertexStart
This variable will be used when adding the vertices to at each branch node to aid in the building of branch segment indices. It will contain the vertex index where the current node's ring of vertices begins in the branch mesh. Remember that each branch will be a separate mesh, so this value will be the index where the ring of vertices starts in the branch mesh's vertex list. We will need to know this when adding branch segments to our mesh.

A branch segment is a cylinder of faces formed from the vertices at the parent node and the vertices at this node. To build these indices we will need an easy way of knowing the position at which a node's ring of vertices is placed in the branch mesh's vertex buffer. If a node had this value set to 50, and the branch resolution property of the tree was set to 8, we would know when generating the indices that this node's vertices will be positioned at 50 through 57 in the branch mesh's vertex buffer. This is a value we will generate when adding the vertices to each branch mesh in the second phase of creation.

ULONG UID
Every branch node in the virtual tree will contain a unique ID that identifies the node. This will start off at a value of zero for the first node of the hierarchy and will be incremented for each new node generated. The UID of the N^{th} node created will simply be N-1 because we start at 0 for the root node. So if the UID of a branch node had a value of 45, this would mean it was the 46^{th} branch node created during the virtual tree creation process. Why would we need to know this?

We will need this information when we build the skeleton for the actor in the second phase of tree creation. Certain nodes in the virtual tree hierarchy will become bones in the actor's frame hierarchy and as we know, frames that we wish to animate must be assigned names. We also need frames to have names in order to set up the skinning information. When a branch node is determined to be a candidate for a bone, we will need to give that frame a name. We also need to know that the name we assign the frame is unique with respect to any other frames in the hierarchy. We will assign a name to each frame using the format Branch_N where N is the UID of the branch node from which the bone is being created. Therefore, if we determine during the creation of the actor's skeleton that a branch node with a UID of 231 is to be used to create a bone for the actor's frame hierarchy, the name given to that frame will be Branch_231, which is guaranteed to be unique from any other frames we add.

bool BoneNode
When we discussed the TreeGrowthProperties structure, we examined a member called Bone_Resolution. By default it was set to 3. We discussed how this defines a ratio describing the number of bones we should create compared to the number of branch nodes in our virtual tree. The default value 3 means that we will create a bone for every third node in the branch node hierarchy. When building the actor's frame hierarchy (after the virtual tree of branch nodes has been constructed), we will traverse the branch node hierarchy starting at the root node. The Branch Begin node of every branch will always have a bone created for it. From that point on, we will create a bone using every third

node in the branch. Branch End nodes will never have a bone constructed from it, as it makes little sense to stick a joint at the tip. This second phase is where the frame hierarchy will be constructed.

The bones that we create from branch nodes between the Branch_Begin and Branch_End nodes form the skeleton for that mesh. Remember, each branch is a separate mesh and therefore will have its own bones connected into a larger hierarchy of bones for the entire actor. That is, the actor will have a skeleton that will represent the bones of the entire tree, but any given branch mesh will only use a localized set of those bones.

After the branch node hierarchy has been constructed in the initial phase, we can traverse that hierarchy and easily calculate which branch nodes should become bones and add them to the actor's frame hierarchy. This second phase is where the actor's frame hierarchy gets constructed from the virtual tree. Whenever we decide to make a bone from a branch node, we will set its BoneNode boolean member to true. This will be useful later on when building the skins for the various branches because we will need the original information stored in the branch node to calculate the bone offset matrix for a given bone in the hierarchy. It allows us to easily traverse a branch and find the branch nodes that have been used to construct bones.

As the BranchSegment member of a branch node contains the zero based index of that node within a given branch (a mesh), we can easily determine if a branch node will be a bone during the building of the actor's frame hierarchy. Simply put, every time the modulus of the branch node's BranchSegment member and the BoneResolution member equals zero, we have skipped the correct number of bones and it is time to mark the next new node as a bone node and create the accompanying frame in the hierarchy. Example code to determine if a newly created node (pNode) is a bone node is shown below.

```
if ( pNode->Type == BRANCH_BEGIN ||
   ((pNode->BranchSegment % m_Properties.Bone_Resolution)==0 && pNode->Type!=BRANCH_END))
{
        pNode->BoneNode = true;
        …
        …
}
```

As you can see, if this is a Branch_Begin node, then it is a node that starts a new branch and will become the first (root) bone of that branch mesh. In such a case we always make it a bone node. Otherwise, the only time we make a node a bone node is when the modulus of the local branch index of the node and the bone resolution wraps back around to zero (i.e., we have skipped the correct number of nodes). Notice as well that this is only true if the node in question is not a Branch_End node.

LPD3DXFRAME pBone

This member is also used during the creation of the actor's frame hierarchy in the second phase, and as such, is related and set using the same conditions as described for the previous member. As discussed, after building the virtual tree, we will enter the second phase where we construct the frame hierarchy (the skeleton) of the actor. We will traverse the branch node hierarchy and determine that certain nodes are bone nodes. For each one that we find, we will allocate a new D3DXFRAME, calculate its relative matrix based on the information stored in the branch node, and assign the frame (bone) a name before attaching it to the actor's hierarchy. We will then assign the new frame pointer to the source branch node's pBone member and set the branch node's BoneNode boolean to true. This establishes a

connection between the frame we have just added to the hierarchy and the branch node it was originally created from. This connectivity information will be used later when we manually build the ID3DXSkinInfo object for the branch mesh.

We have now discussed all there is to discuss about the structures used by CTreeActor and the properties that the tree generation process will use. While this has been an awful lot to take in without any code to look at, it is important that you understand the process overall before viewing the highly recursive code.

12.4 Source Code Walkthrough - CTreeActor

It is now time to look at our CTreeActor class, which is derived from CActor. Below you can see how the class is defined in CTreeActor.h. We will discuss the various new member variables of the actor in a moment. We will not immediately discuss the various methods of the class now as we will cover each one when we encounter it in the general flow of explaining the tree creation process.

```
class CTreeActor : public CActor
{
public:

    enum BranchNodeType { BRANCH_BEGIN = 1,BRANCH_SEGMENT = 2,BRANCH_END = 3 };

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


    // Public Functions for This Class
    void                 SetGrowthProperties (const TreeGrowthProperties & Prop );
    TreeGrowthProperties GetGrowthProperties ( ) const;
    void                 SetBranchMaterial ( LPCTSTR strTexture,
                                             D3DMATERIAL9 * pMaterial = NULL );

    HRESULT   GenerateTree( ULONG Options,
                            LPDIRECT3DDEVICE9 pD3DDevice,
                            const D3DXVECTOR3 & vecDimensions,
                            const D3DXVECTOR3 & vecInitialDir
                                              = D3DXVECTOR3( 0.0f, 1.0f, 0.0f ),
                            ULONG BranchSeed = 0 );


    HRESULT                 GenerateAnimation   ( D3DXVECTOR3 vecWindDir,
                                                  float fWindStrength,
                                                  bool bApplyCustomSets = true );

    virtual void            Release             ( );

private:

    // Private Functions for This Class
```

```
    HRESULT     GenerateBranches    ( const D3DXVECTOR3 & vecDimensions,
                                      const D3DXVECTOR3 & vecInitialDir
                                          = D3DXVECTOR3( 0.0f, 1.0f, 0.0f ),
                                      ULONG Seed = 0 );
    void        BuildBranchNodes    ( BranchNode * pNode,
                                      ULONG & BranchUID, ULONG Iteration = 0 );

    bool        ChanceResult        ( float fValue ) const;
    void        DeviateNode         ( BranchNode * pNode,
                                      float fAzimuthThetaMin,
                                      float fAzimuthThetaMax,
                                      float fPolarTheta = 360.0f ) const;

    HRESULT     BuildFrameHierarchy ( ID3DXAllocateHierarchy * pAllocate );
    HRESULT     BuildNode           ( BranchNode * pNode,
                                      D3DXFRAME * pParent,
                                      CTriMesh * pMesh,
                                      const D3DXMATRIX & mtxCombined,
                                      ID3DXAllocateHierarchy * pAllocate );

    HRESULT     BuildSkinInfo       ( BranchNode * pNode,
                                      CTriMesh * pMeshData
                                      LPD3DXSKININFO * ppSkinInfo );

    HRESULT     BuildNodeAnimation  ( BranchNode * pNode,
                                      const D3DXVECTOR3 & vecWindAxis,
                                      float fWindStrength,
                                      LPD3DXKEYFRAMEDANIMATIONSET pAnimSet );
    HRESULT     AddBranchSegment    ( BranchNode * pNode, CTriMesh * pMesh );


   // Private Variables for This Class
    ULONG                   m_nBranchSeed;
    TreeGrowthProperties    m_Properties;
    BranchNode             *m_pHeadNode;

    D3DMATERIAL9            m_Material;
    LPTSTR                  m_strTexture;
};
```

While the list of new member variables we have added is extremely small (only five), do bear in mind at all times that this class is derived from CActor and thus has access to all of its member variables and functionality as well. That is why you cannot see (for example) a member variable that points to the root frame of the tree's frame hierarchy. As you know, the frame hierarchy member variables (and the method to traverse them) are all inherited from the base class (CActor). Let us now discuss these new member variables one at a time.

ULONG m_nBranchSeed

As you certainly aware by now, the assembling of our tree will call for a lot of randomly generated numbers. For example, we will generate random angles for direction vector deviations between nodes in a branch and we will also use random numbers to decide whether or not a branch node currently being processed should spawn a new child branch. You will see when we cover the code in a moment that random number generation is used literally throughout the entire tree generation process.

We generate random numbers using the rand function (part of the standard C runtime libraries). Of course, a computer cannot generate a truly random number since this is essentially like asking the computer to make a real choice. Only sentient beings that are self aware can make random choices. So, the rand function (and its sibling function, srand) just generates and returns numbers using a simple calculation that returns a string of numbers that seem random to a human being. All the rand function is actually doing is performing a calculation where the srand function is used to set an initial value (a seed) that is used in that calculation. Each time the rand function is called, the initial seed value we set is updated to new value which is then used to influence the value returned in the next rand call. For example, lets us imagine that the C runtime library stores some global value called 'next' which is initially set to 1. That is, the default seed for the rand function is 1.

```
unsigned long int next = 1;

/* rand: return pseudo-random integer on 0..32767 */
int rand(void)
{
    next = next * 1103515245 + 12345;
    return (unsigned int) (next/65536) % 32768;
}
```

As you can see, the rand function does not generate anything randomly. The first time it is called the 'next' variable will be set to the seed value (1 by default). The rand function simply multiplies this value by a very large number. Notice however that the 'next' value is updated with the result so that each time the rand function is called a different value will be returned. It then simply returns the number divided by 65536 and has a modulus performed with 32768 to snap the value into the correct range of an unsigned integer.

We use the srand function to seed the C random number generator (i.e., change the initial value of the 'next' variable in the above code). What is vitally important to note is that this list of pseudo-random numbers generated will *never* change. So if we use the same seed at the start of our application, the random numbers generated will be exactly the same every time the application is run. After all, every time the application is run, the same starting point in that list of random numbers is used. Thus ten calls to the rand function will generate the same ten random numbers every time the application is run. In terms of our tree, this means regardless of how many times we use the rand function in our generation process, our GenerateTree function will create the same tree each time the application is run. This may or may not be desirable.

Often it is not desirable. If a game uses random numbers to generate random events, we do not want these events to happen on cue every single time at the exact same place every time the application is run. Therefore, usually the value used to seed the random number generator is taken from a value that is unique every time the application is run (e.g., the timestamp of the current system time or the number of microseconds that have passed since the computer was switched on). We take such a value and use the srand function to seed the random generation process. Below we can see that the call to srand simply sets the initial value of the 'next' value used in the rand calls. If we use a different seed every time the application runs (system time for example), the random numbers generated will be different every time also.

```
/* srand: set seed for rand() */
void srand(unsigned int seed)
{
    next = seed;
}
```

That fact that using the same seed will generate the same string of pseudo-random numbers can actually be beneficial in our tree creation process. When the application calls the CTreeActor::GenerateTree method we can optionally pass in a seed value that will be used to seed the random number generator prior to tree creation. We will use the timeGetTime function call to seed the random number generator if the application passes no seed. As this value will change each time a tree is generated, we will get different trees every time the function is called. However, you also have the option to pass in any seed value that you like so that the same tree is generated. This allows you to continually generate trees using different seed values until you find a tree that you really like. You can then remember those seeds and use them to generate the trees for your game, knowing exactly what the trees that will be generated will look like.

The m_nBranchSeed member of the class will contain the seed that was used to generate the tree. This will be either the seed value passed in by the application or the value returned from timeGetTime. If you see a tree that was randomly generated that you really like, you can examine the contents of this member and use this seed in the future to create that same tree.

TreeGrowthProperties m_Properties
This member contains the creation properties used to generate the tree. We set the members of this structure using the CTreeActor::SetGrowthProperties and can also retrieve it using the CTreeActor::GetGrowthProperties methods. We have already discussed all the members of this structure in detail..

BranchNode *m_pHeadNode
This member stores a pointer to the root branch node of the virtual tree hierarchy that was generated during tree creation. Remember, this is not the root frame of the actor's hierarchy (stored in a base class member variable). The hierarchy of branch nodes is used to grow a virtual tree representation and then used to build the frame hierarchy and meshes of the actor.

D3DMATERIAL9 m_Material
LPTSTR m_strTexture
Our branches will need to have a texture and material applied to them. We set these values using the CTreeActor::SetBranchMaterial function. Notice that we do not pass in an actual texture, only the filename. Essentially, this is the same information our actor is passed by D3DX when we load a material from an X file. This is certainly by design because from the CActor's perspective, there will be no difference. What do we mean by this?

After we have generated our virtual tree we will step through that hierarchy and generate the frame hierarchy similarly to how D3DX does it when we call D3DXLoadMeshHierarchyFromX. Every time we wish to allocate a new frame for example, we will call the actor's CAllocateHierarchy::CreateFrame method, just as D3DX does during the loading process. Our CAllocateHierarchy object is already written, so we use it again in exactly the same way. Similarly, every time D3DX encounters a mesh

during the loading process, it will call our CAllocateHierarchy::CreateMeshContainer method which is responsible for loading textures and materials (via registered callbacks) and for generating the actual skinned mesh. We will follow this exact same technique so that all the code we have developed in that function can be re-used without modification to generate our tree meshes.

When we traverse the virtual tree hierarchy generating mesh data, we will do two things. We will generate a regular mesh and we will also populate an ID3DXSkinInfo object with bone information. Once we have done that, we will simply call the CAllocateHierarchy::CreateMeshContainer method passing the regular mesh we have generated for the branch being processed and its skin info. We will also pass it the material and the texture filename that has been set for the tree. As we know, the CreateMeshContainer method will generate a skinned mesh from the regular mesh passed in and will also take care of calling any callback functions to handle the loading and processing of textures and materials. From the perspective of CreateMeshContainer, nothing is different from the case where the mesh data is being loaded by D3DX. We still pass it a regular mesh, texture filename and material structures, the same format in which the D3DXLoadMeshHierarchyFromX provides the mesh information.

It is actually quite a complex relationship when you examine it. In our application, our CTreeActor is set to non-managed mode prior to tree generation (the application registers an attribute callback function). As we know, this callback function is responsible for loading textures from filenames and storing those textures and their accompanying materials in the scene database. We also set the texture filename and the material the tree should use immediately afterwards. Next we make the call to the GenerateTree method to generate the actual meshes. So, for each branch mesh we create, the texture filename is passed to the CreateMeshContainer method which then sends that same texture filename back to the application defined callback which loads the actual texture used by the actor. So, the application supplied the actor with a texture filename which is later passed back to its callback function where it will load that texture.

Constructor - CTreeActor

The constructor of CTreeActor simply initializes the tree's growth properties to some default values that will be overridden as soon as the application calls the SetGrowthProperties method. It also sets a default material in case the application does not call the SetBranchMaterial method to specifically set one. The constructor is shown below. Please note that the values assigned to each of the tree growth properties may differ in the actual source code. At the time of this writing, the development team was still experimenting with their preferred defaults.

```
CTreeActor::CTreeActor()
{
    // Call base constructor
    CActor::CActor();

    // Reset all required values
    m_nBranchSeed = 0;
    m_pHeadNode   = NULL;
    m_strTexture  = NULL;
```

```
    // Setup some useful initial growth properties
    m_Properties.Max_Iteration_Count        = 21;
    m_Properties.Initial_Branch_Count       = 1;
    m_Properties.Min_Split_Iteration        = 2;
    m_Properties.Max_Split_Iteration        = 20;
    m_Properties.Min_Split_Size             = 0.8f;
    m_Properties.Max_Split_Size             = 20.0f;

    // 25% Chance of splitting in two
    m_Properties.Two_Split_Chance           = 15.0f;

    // 25% Chance of splitting in three
    m_Properties.Three_Split_Chance         = 5.0f;

    // 1% Chance of splitting into four
    m_Properties.Four_Split_Chance          = 1.0f;

    // 5% Chance that a branch will end when a split occurs.
    m_Properties.Split_End_Chance           = 5.0f;

    m_Properties.Segment_Deviation_Chance   = 60.0f;    // 60% chance of deviation
    m_Properties.Segment_Deviation_Min_Cone = 0.0f;     // Min Cone Angle
    m_Properties.Segment_Deviation_Max_Cone = 30.0f;    // Max Cone angle
    m_Properties.Segment_Deviation_Rotate   = 10.0f;    // Max Polar rotation

    m_Properties.Length_Falloff_Scale       = 0.1f;     // Segment Length falloff
                                                        // happens 10 times slower
                                                        // than branch thickness
                                                        // fall off

    // Deviation properties for new child branch
    m_Properties.Split_Deviation_Min_Cone   = 10.0f;    // Min Cone Angle
    m_Properties.Split_Deviation_Max_Cone   = 70.0f;    // Max Cone Angle
    m_Properties.Split_Deviation_Rotate     = 360.0f;   // Max Polar rotation

    m_Properties.SegDev_Parent_Weight       = 0.0f;     // Weight with which the
                                                        // original direction of
                                                        // the segments parent is
                                                        // averaged with the
                                                        // deviation

    m_Properties.SegDev_GrowthDir_Weight    = 0.4f;     // Weight with which the
                                                        // growth direction vector
                                                        // is averaged with the
                                                        // deviated segment
                                                        // direction

    m_Properties.Branch_Resolution          = 8;        // Number of vertices used
                                                        // for each branch mesh
                                                        // segment ring

    m_Properties.Bone_Resolution            = 3;        // One bone every 3 nodes

    m_Properties.Texture_Scale_U            = 1.0f;
    m_Properties.Texture_Scale_V            = 1.0f;
```

```
    // Growth direction of tree
    m_Properties.Growth_Dir                 = D3DXVECTOR3( 0.0f, 1.0f, 0.0f );

    // Setup default material properties
    m_Material.Diffuse   = D3DXCOLOR( 0.8f, 0.8f, 0.8f, 0.8f );
    m_Material.Emissive  = D3DXCOLOR( 0.0f, 0.0f, 0.0f, 1.0f );
    m_Material.Ambient   = D3DXCOLOR( 0.0f, 0.0f, 0.0f, 1.0f );
    m_Material.Specular  = D3DXCOLOR( 0.0f, 0.0f, 0.0f, 1.0f );
    m_Material.Power     = 0.0f;

}
```

What is important to note is that the first thing we do in this constructor is issue a call to the base class constructor so that CActor can perform its own initialization. Remember, the base class sets a default skinning method and initializes the actor's callback array. It also sets some default parameters for the animation controller.

Destructor – CTreeActor

Like all of our destructors, this destructor is also very simple due to the fact that it hands off the clean-up to its Release method (which can also be called by the application to force a clean up). What we do release in this function is any texture filename string that may exist.

When we look at the CTreeActor::SetBranchMaterial method in a moment, you will see that the texture filename passed in is duplicated and stored in the m_strTexture member variable (using _tcsdup). Therefore, we must make sure on destruction that we clean that memory up in the appropriate way. We then issue a call to the CTreeActor::Release method to do any final clean up.

```
CTreeActor::~CTreeActor()
{
    // Release any allocated memory
    Release();

    // Release the properties.
    if ( m_strTexture ) free( m_strTexture ); // Allocated by _tcsdup

    // Reset variables
    m_strTexture = NULL;
}
```

Release – CTreeActor

The Release method is small because the derived class has very few resources to clean up. What we must do however is call the base class Release method so that the frame hierarchy and meshes managed by the base class are released. We know the CActor base class performs quite a bit of clean up, such as releasing interfaces to animation controllers, destroying CTriMesh objects, and de-allocating the entire

frame hierarchy. Once we have called the base class's Release method, the only additional information that needs to be released in CTreeActor is the virtual tree hierarchy. When we looked at the BranchNode structure earlier, we also saw how its constructor took care of deleting its sibling and child pointers. Therefore, all we have to do is delete the root branch node (m_pHeadNode), which will delete its child, which will delete its child, and so on, causing a traversal and deletion of the entire branch node hierarchy.

```
void CTreeActor::Release()
{
    // Call base release
    CActor::Release();

    // Destroy the tree hierarchy data
    if ( m_pHeadNode  ) delete m_pHeadNode;

    // Clear variables.
    m_pHeadNode  = NULL;
}
```

SetGrowthProperties / GetGrowthProperties – CTreeActor

One task that the application will want to perform before generating the actual tree, is setting the growth properties that will influence the tree generation process. This function takes as its only parameter a TreeGrowthProperties structure whose values will be stored in the CTreeActor's m_Properties member. We also supply a function for retrieving the properties of a CTreeActor. Both the 'Set' and 'Get' functions are shown below.

```
void CTreeActor::SetGrowthProperties( const TreeGrowthProperties & Prop )
{
    // Store the properties
    m_Properties = Prop;
}
```

```
TreeGrowthProperties CTreeActor::GetGrowthProperties( ) const
{
    // Return the properties
    return m_Properties;
}
```

It is important that you set the growth properties of the tree prior to calling the GenerateTree method since it uses the values stored in the m_Properties structure to influence tree generation. Setting the properties after the tree has been generated will have no effect.

SetBranchMaterial – CTreeActor

Another task that you will probably want to perform before calling GenerateTree is informing the tree about the texture and material you would like to have applied to its mesh faces. We do this using the CTreeActor::SetBranchMaterial function. The function accepts two parameters. The first should be a string containing the filename of the texture you wish to use and the second parameter should be a pointer to a D3DMATERIAL9 structure containing the light reflectance properties that should be applied to the faces of the tree.

```
void CTreeActor::SetBranchMaterial( LPCTSTR strTexture, D3DMATERIAL9 * pMaterial )
{
    // Free any previous texture name
    if ( m_strTexture ) free( m_strTexture );
    m_strTexture = NULL;

    // Store the material
    if ( pMaterial ) m_Material = *pMaterial;

    // Duplicate the texture filename if any
    if ( strTexture ) m_strTexture = _tcsdup( strTexture );
}
```

If the tree already has a texture name set, then its memory is released. The material passed is then copied into the m_Material member variable, replacing the default values we assigned to it in the constructor. Finally, we use the _tcsdup function to make a copy of the texture filename. A pointer to this copy is stored in the m_strTexture member variable.

GenerateTree - CTreeActor

We are now ready to examine the GenerateTree method, which is invoked by the caller to build the entire tree. We can think of this method as being the replacement to the regular actor's LoadActorFromX method in that on function return, the actor will have a fully populated frame hierarchy (and potentially even some animation data). This function is actually just a front end function for the caller since the tree building mechanism is mostly done through helper functions called from this method. We will examine each of the helper functions one at a time afterwards. However, by looking at the GenerateTree function first, we will be able to see the overall order in which the various processes are invoked.

GenerateTree accepts five parameters (the final two are optional). The first parameter is a combination of one or more D3DXMESH flags which instruct the tree in which of the device's resource pools the vertex and index buffers for the branch meshes should be allocated. The second parameter is a pointer to the device that will essentially own the actor and its meshes. The third parameter is the initial dimensions for the root node in the branch node hierarchy. For example, passing a vector of (2, 3, 6) means the thickness of the root branch segment is defined as an ellipse that has radii X=2 and Y=3 in world space. These values are used to place the ring of vertices at the root branch node of the correct

size. The size of each branch segment will get smaller as we step along the branch. That is why we only pass in the size of the root branch node which will then be automatically downsized in each recursive step. The Z component of this vector describes the length of the root branch segment. Using the vector specified above, this means after the root node has been inserted, we will move a distance of 6 units along the root node's direction vector before placing the next node in that branch. As both these nodes define the bottom and top of the same cylinder, this Z value really defines the length of the root branch segment. The fourth (optional) parameter is another 3D vector describing the initial direction vector of the root node of the virtual tree. If we were to pass a vector of <1,0,0> the initial segment of the root branch would grow horizontally along the X axis (e.g., out of a cliff face). If the growth vector of the tree has been set (in the tree properties) to <0,1,0>, the tree would start to grow upwards as each new node was added. If you omit this parameter, a default vector of <0,1,0> will be used. This is a very sensible default since you will usually want your tree to grow vertically upwards. The final (optional) parameter is an unsigned long integer that will be used to seed the random number generator. If omitted, the current system time will be used to seed the random number generator.

In the first section of the function, we allocate an instance of our CAllocateHierarchy class. This will be used later so that we can call its CreateFrame and CreateMeshContainer methods to generate the frames of the hierarchy and the skinned meshes. We return if a valid device was not passed.

```
HRESULT CTreeActor::GenerateTree( ULONG Options, LPDIRECT3DDEVICE9 pD3DDevice,
                                  const D3DXVECTOR3 & vecDimensions,
                                  const D3DXVECTOR3 & vecInitialDir ,
                                  ULONG BranchSeed )
{
    HRESULT            hRet;
    CAllocateHierarchy Allocator( this );

    // Validate parameters
    if ( !pD3DDevice ) return D3DERR_INVALIDCALL;

    // Release previous data.
    Release();

    // Store the D3D Device here
    m_pD3DDevice = pD3DDevice;
    m_pD3DDevice->AddRef();

    // Store options
    m_nOptions = Options;
```

Provided the device is valid we call the CTreeActor::Release method to release any previous data that may have been generated for this tree object. This allows us to use the same object to generate another tree simply by calling its GenerateTree method again. Remember, the Release method also passes the request to the base class Release method, ensuring that all frames, mesh, and attribute data is released before generating a new tree from scratch. Next we make a copy of the device interface, increment the interface's reference count, and store the mesh creation options in the m_nOptions member.

The final section of the function code builds the entire tree through the invocation of three other methods, each of which is responsible for one phase of the tree generation. We will look at the code to each of these three functions as we go along.

```
    // Generate the branches
    hRet = GenerateBranches( vecDimensions, vecInitialDir, BranchSeed );
    if (FAILED(hRet)) return hRet;

    // Build the frame hierarchy
    hRet = BuildFrameHierarchy( &Allocator );
    if (FAILED(hRet)) return hRet;

    // Build the bone matrix tables for all skinned meshes stored here
    if ( m_pFrameRoot )
    {
        hRet = BuildBoneMatrixPointers( m_pFrameRoot );
        if ( FAILED(hRet) ) return hRet;

    } // End if no hierarchy

    // All is well.
    return D3D_OK;
}
```

The GenerateBranches function is called first and completes the first phase of the tree building process. It is responsible to building the branch node hierarchy that contains our virtual tree representation. When the GenerateBranches method returns, we will have a virtual tree hierarchy, the root branch node of which will be pointed to by the m_pHeadNode member variable. At this point, the actor will not yet have had its frame hierarchy built and no mesh data will have been generated. That is what the next function BuildFrameHierarchy does. It accomplish phase two of the tree generation process.

BuildFrameHierarchy is responsible for building the actor's D3DXFRAME hierarchy and generating all skinned meshes. It does so by traversing the virtual tree hierarchy assembled in phase one and inserting vertices and frames for the relevant branch nodes. Once the BuildFrameHierarchy function returns, the actor's frame hierarchy will have been created and all the skinned meshes will have been created and attached to that hierarchy as well. However, at this point there is still one final step to perform which you should recall from the last chapter. During the building of the frame hierarchy, we cannot store the bone pointers needed for our mesh containers until the frame hierarchy has been created in its entirety. Therefore, just as we do in the CActor::LoadActorFromX function, after the frame hierarchy has been created, we traverse it and calculate the absolute matrices at each frame (the bone matrices) before storing all the bone matrices used by a given mesh container in its bone matrix pointer array. We looked at the code to the CActor::BuildBoneMatrixPointers function in the previous chapter's workbook.

Now we are ready to look at the first function called in the above code which implements phase one (i.e., the building of the virtual tree branch node hierarchy).

GenerateBranches - CTreeActor

This function is called from the GenerateTree method and is passed the dimensions of the root branch node, the direction of the root branch node, and (optionally) the random seed. The first thing the function does is seed the random number generator. The default value of the Seed parameter is zero if omitted from the parameter list, so the first thing we do if this is the case is set the seed value to the value returned from the timeGetTime function. We then store either the passed seed value or the seed value just calculated in the m_nBranchSeed member variable. Finally, we seed the random number generator via a call to srand.

```
HRESULT CTreeActor::GenerateBranches( const D3DXVECTOR3 & vecDimensions,
                                      const D3DXVECTOR3 & vecInitialDir ,
                                      ULONG Seed )
{
    ULONG i;
    ULONG BranchUID = 0;

    // Seed the random number generator with current time if not specified
    if ( Seed == 0 ) Seed = timeGetTime();

    // Store the seed value
    m_nBranchSeed = Seed;

    // Seed the generator
    srand( Seed );
```

This function is responsible for creating the root node of the branch node hierarchy. It will then pass this node to the BuildBranchNodes function (a recursive function that will repeatedly call itself to create new branch nodes for the hierarchy). However, as mentioned earlier, we do have the option of creating a tree that has multiple initial branches (multiple trunks), so we may have multiple nodes in the root level of the tree. While we will usually expect a value of 1 to be stored in the Initial_Branch_Count member of the tree's growth properties structure, this may not always be the case. Therefore, we will set up a loop that counts up to the value stored in the Initial_Branch_Count member and create a new branch node for each root branch. As we may be creating multiple branch nodes at the root level (one for each initial branch) we will connect these as siblings at the root level. Each root branch node will be passed into the BuildBranchNodes recursive function to generate the entire tree for that branch recursively.

In the next section of code we show the start of that loop. Each iteration allocates a new branch node. If a previous node has been created in a previous iteration then we attach the new branch node to the previous node's sibling pointer. If not, then this is the first node we have generated and will assign the m_pHeadNode member variable to point at this node. In other words, if the initial branch count was set to 3, in the first iteration we would generate a new branch node and assign it to the m_pHeadNode pointer. In the second iteration we would create a new branch node and assign it to the head node's sibling pointer. In the third and final iteration we would create a new branch node and assign it to the sibling pointer of the node we created in loop iteration 2, and so on. Thus, we are creating a sibling list of Branch_Begin nodes at the root level of the hierarchy.

```
    BranchNode * pPrevNode = NULL;

    // Generate the required set of head branches
    for ( i = 0; i < m_Properties.Initial_Branch_Count; ++i )
    {
        BranchNode * pNewNode = new BranchNode;
        if ( !pNewNode ) continue;

        // Store in the appropriate place
        if ( pPrevNode )
            pPrevNode->Sibling = pNewNode;
        else
            m_pHeadNode = pNewNode;

        // Setup the node
        pNewNode->UID        = BranchUID++;
        pNewNode->Type       = BRANCH_BEGIN;
        pNewNode->Direction  = vecInitialDir;
        pNewNode->Dimensions = vecDimensions;
        pNewNode->Iteration  = 0;
```

In the above code, you can see that for each root node we create we assign it a unique branch node ID (BranchUID). This value was set to 0 at the start of this function and is incremented with each new node we add. We also know that any branch nodes we create at the root level will be the start nodes of trunk branches, so we set the type of each branch node created in this loop to the type Branch_Begin. When building the meshes later, we will know that each of these nodes starts a new branch mesh. We also store the direction vector and the dimension vector passed into the function in the node. Finally, we set the iteration of the nodes to zero. This iteration will be passed down the branches and incremented at each level. Every branch node added at the root level however will be set to zero since they exist at the same level in the hierarchy and for all intents and purposes are the start nodes of separate trees linked at the root level. The position of each node is not set here as this will have been set to 0 in the constructor of the branch node structure. We wish each trunk branch to be positioned at (0,0,0) in tree space.

We know the direction vector of the nodes we are creating because it was passed into the function. However, we learned earlier that each node also needs a right vector. The right vector of the root node(s) will be passed down the tree and modified at each node so that it remains correctly orthogonal to the node's deviated direction vector. We do not require the application to pass in the right vector of the root node(s) as we can calculate that easily. All we have to do is find the coordinate system axis (X, Y or Z) that is least aligned to the direction vector. This will allow us to find a vector that is definitely not the same as the direction vector. We find this axis simply by taking the absolute values of the direction vector's components and searching for the smallest component. For example, we know that if the direction vector has a Z component of 0, then we can use the Z axis of the coordinate system since this is definitely not close to being aligned to the direction vector. All we are doing here is safely picking an axis which is not the direction vector. Once we have that axis vector, we can perform the cross product between the direction vector and that axis vector to create the right vector for the node. The following code shows how we find this axis vector and cross it with the direction vector to generate the right vector. The right vector is then stored in the branch node.

```
        // Get absolute normal vector
        float x = fabsf( pNewNode->Direction.x );
```

```
        float y = fabsf( pNewNode->Direction.y );
        float z = fabsf( pNewNode->Direction.z );
        float fNorm = x;

        // Find the best vector to use as right vector
        D3DXVECTOR3 vecCross = D3DXVECTOR3( 1.0f, 0.0f, 0.0f );
        if ( fNorm > y )
        { fNorm = y; vecCross = D3DXVECTOR3( 0.0f, 1.0f, 0.0f ); }

         if ( fNorm > z )
        { fNorm = z; vecCross = D3DXVECTOR3( 0.0f, 0.0f, 1.0f ); }

         D3DXVec3Cross( &pNewNode->Right, &pNewNode->Direction, &vecCross );
```

At this point we have successfully populated the new branch node with its information. However, if we are generating multiple branches at the root level (Initial_Branch_Count > 1) then we cannot simply assign the same direction and right vectors to each branch node; otherwise, each root branch node we create will be placed in the exact same position and with the same orientation as the other nodes in its sibling list. So, if we are only creating a single node, then our job is done and we pass this node into the BuildBranchNodes function to start the recursive branch building process. The branch node will have the exact direction vector that we passed into the function. However, if we are creating multiple root branch nodes, then we must deviate the direction vector of each node so that they are slightly different. This will allow us to generate a tree where multiple trunks sprout from the same location but with different directions. If deviation is needed, we call the DeviateNode function to perform that deviation. This function (which will be covered in a moment) is passed the node, the min and max cone angles and a rotation angle. This will deviate the vector a random angle about its right vector with a range of –180 to +180 degrees. It will also rotate the new deviated direction vector a random angle of 180 degrees about the original direction vector. Finally, it will rotate the node's right vector so that it stays orthogonal to the new deviated direction vector.

We covered how vector deviation is performed in theory earlier on in the chapter, so refer back if you need to brush up. After the DeviateNode function returns, we have modified the root node direction and right vectors and are ready to pass this node into the BuildBranchNodes function. This function is a recursive process that repeats until all the branch segments of the root node and any child branches it spawns are calculated. Suffice to say, when BuildBranchNodes returns program flow back to this function, the entire virtual tree for the current root node we are processing will have been created.

```
        // Deviate if we're starting with more than one branch
        if ( m_Properties.Initial_Branch_Count > 1 )
             DeviateNode( pNewNode, 0.0f, 180.0f, 180.0f );

        // Start the recursive build for this branch
        BuildBranchNodes( pNewNode, BranchUID );

        // Store previous node
        pPrevNode = pNewNode;

    } // Next Initial Branch
```

At the end of the loop shown in the above code, we assign the pPrevNode pointer to point at the node we have just generated. This way, in the next iteration (assuming multiple roots) of the loop, we have access to the node generated in the previous iteration so that we can add a new root node to its sibling list.

When the loop finishes, the entire virtual tree hierarchy will have been created. We will examine the BuildBranchNodes recursive function in detail shortly, as this is where all of the work happens for phase one.

In the next and final section of the code we test to see that the root node of the tree is valid and if not we return E_OUTOFMEMORY. If this is NULL, then the very first root node could not be allocated so we must return failure. Otherwise, we return success.

```
    // Fail if we failed.
    if ( !m_pHeadNode ) return E_OUTOFMEMORY;

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

There are three functions called from the previous function that we need to discuss. The first one called is DeviateNode. It will be used many times throughout the tree building process to randomly deviate the direction vector of a child node from that of its parent node. Let us have a look at that function next.

DeviateNode - CTreeActor

DeviateNode is a small function that is called many times throughout the tree generation process to randomly deviate the vector of a child node from that of its parent over some given a range. It takes four parameters. The first parameter is the node whose direction vector we wish to deviate. The direction vector of a newly created node that is passed into this function will initially have its direction vector inherited from the parent node (or in the case of the GenerateBranches function, explicitly set to the direction vector passed by the application). Therefore, there is no reason to also pass in the parent node's vectors (direction and right) which we wish to rotate around, as initially, the vectors stored in the child node we pass will be equal to its parent vectors.

The second and third parameters specify the minimum and maximum deviation angles in degrees. This is the amount we wish to rotate the direction vector clockwise or counter-counter clockwise about the parent node's right vector. This forms a minimum and maximum cone of rotation. We can initially perform this rotation of the direction vector about the node's right vector, since at this point the right vector of the node will be equal to the right vector stored in its parent node. The final parameter is the maximum polar rotation angle we wish to rotate the (now rotated) direction vector around the parent's direction vector. Passing a value of 180 degrees will allow us to swing the direction vector to any position around the branch (90 degrees left or right from the rotated position). Here is the first section of the code followed by an explanation.

```
void CTreeActor::DeviateNode( BranchNode * pNode,
                              float fAzimuthThetaMin,
                              float fAzimuthThetaMax,
                              float fPolarTheta ) const
{
    // Cache direction
    D3DXVECTOR3 vecNormal = pNode->Direction;

    // Generate Azimuth / Polar angles
    float fAzimuth = (float)rand() / (float)RAND_MAX;
    float fPolar   = (float)rand() / (float)RAND_MAX;
    fAzimuth = fAzimuthThetaMin +
               ((fAzimuthThetaMax - fAzimuthThetaMin) * fAzimuth);

    fPolar   = (fPolarTheta * fPolar) - (fPolarTheta / 2.0f);
```

First we make a copy of the node's direction vector. This is important as this currently holds the direction vector of the parent node. Once we rotate this vector it will be overwritten and we will have lost our ability to rotate the rotated direction vector about the original direction vector of the parent.

We then generate two random rotation values for the fAzimuth and fPolar local variables. The rand function generates a random integer in the range [0, RAND_MAX], so dividing the result of rand by RAND_MAX gives a random float in the range [0.0, 1.0]. We calculate the fAzimuth angle using the [0.0, 1.0] random number to find the angle between the minimum cone angle and the maximum cone angle using interpolation. That is, to calculate the first rotation angle (around the parent's right vector), we the minimum cone angle with the difference between the min and max cone angles multiplied by our random float.

We now have an angle in degrees between the minimum and the maximum cone angle. Next we calculate the polar rotation angle. The 0.0 to 1.0 range random number represents an angle between 0 and the maximum polar rotation angle passed into the function. Therefore, scaling the maximum polar rotation angle by the random float will produce an angle value in degrees, in the range [0.0, Max Polar Rotation Angle]. If the passed polar rotation limit was 180 degrees for example, this means we wish to map it into the [–90, +90] range so that we have a random chance of rotating the direction vector clockwise or counter clockwise. Therefore, we subtract from the angle the maximum angle passed divided by two. This maps a value of (Polar Angle / 2) to a zero degree rotation, a value of 0 to –(Polar Angle/2) and a value of Polar Angle to (Polar Angle / 2). This is exactly what we want.

Now that we have our two rotation angles, we need to decide whether the first rotation angle (fAzimuth) is going to rotate the direction vector of the node clockwise or counter-clockwise around the parent's right vector. The following code uses a member function called ChanceResult to generate a number between 1 and 100. This function returns true if it generated a random number smaller than the value passed in (typically a value in the range of 0 to 100 also). We use this function frequently through the tree generation process to decide whether a branch node should be deviated or whether it should spawn child branches. Because we want an equal probability for each direction, we pass in a value of 50. The function will return true if it generates a random number between 0 – 49, otherwise, it will return false. As you can see, the parameter we pass into this function is really a probability value; the lower the value we pass in, the less chance the function has of returning true.

```
// Deviate in both directions to prevent a tendency to lean, whilst still
    // providing support for our 'dead zone' of rotation.
    if ( ChanceResult( 50 ) == true ) fAzimuth = -fAzimuth;
```

If the function returns true, we negate the rotation angle so that rotation happens in the opposite direction.

Now we have two final rotation angles, and it is time to perform the rotations. In the first step, we build a rotation matrix that will rotate vectors about the node's right vector (currently equal to the parent's right vector) by fAzimuth degrees. We then rotate the node's direction vector about the right vector by transforming it using this matrix. This step rotates the local Z axis of the new node into either the left or right semi-circle surround the parent node direction vector.

```
    // Rotate the normal
    D3DXMATRIX mtxRot;
    D3DXMatrixRotationAxis( &mtxRot, &pNode->Right, D3DXToRadian( fAzimuth ) );
    D3DXVec3TransformNormal( &pNode->Direction, &vecNormal, &mtxRot );
```

The first rotation has been performed and our node's direction vector will now be rotated either clockwise or counter-clockwise about the parent's right vector. Next we want to perform the polar rotation. In order to do this we must rotate the node's direction vector around the parent's direction vector (which is why we made a copy in vecNormal at the start of the function) by an angle of fPolar degrees. We build a matrix that performs this rotation and multiply both the node's direction vector and right vector by this matrix. This rotates the direction vector into its final position and also rotates the right vector by the same amount such that it remains orthogonal to the direction vector.

```
    D3DXMatrixRotationAxis( &mtxRot, &vecNormal, D3DXToRadian( fPolar ) );
    D3DXVec3TransformNormal( &pNode->Direction, &pNode->Direction, &mtxRot );
    D3DXVec3TransformNormal( &pNode->Right, &pNode->Right, &mtxRot );
}
```

Finally, we run some vector generation code to ensure that the right vector and direction vector are completely orthogonal. We discussed a similar technique in Chapter Four of Module I when dealing with vector re-generation for the camera class. Because we are passing a direction and right vector down through the recursive process and are repeatedly applying rotations to it, without this next step, floating point accumulation errors would build quite quickly and we would lose orthogonality. Since these vectors represent our node coordinate system which is used in many places during tree and mesh creation, we must not let this happen.

```
   // Ensure that these new vectors are orthogonal
   D3DXVec3Cross( &vecNormal, &pNode->Direction, &pNode->Right );
   D3DXVec3Cross( &pNode->Right, &vecNormal, &pNode->Direction );
   D3DXVec3Normalize( &pNode->Right, &pNode->Right );
```

ChanceResult - CTreeActor

Before continuing to cover the other functions called by GenerateBranches, let us have a look at the ChanceResult function which was called from DeviateNode (and called from various other places throughout the tree generation process).

The function generates a value between 0.0 and 1.0 by dividing the value returned from rand by RAND_MAX. We then multiply this float by 100 so that we have a value between 0.0 and 100.0. If this value is smaller than the probability value passed in (itself a value between 0.0 and 100.0) the function returns true (simulating that the probability has come true in this instance). Otherwise, false is returned.

```
bool CTreeActor::ChanceResult( float fValue ) const
{
    // REALLY simple percentage chance prediction
    if ( (((float)rand() / (float)RAND_MAX) * 100.0f) < fValue ) return true;
    return false;
}
```

BuildBranchNodes – CTreeActor

Recall that after GenerateBranches has deviated the direction vector for each root node (in the multiple root branch case) it passes the root node into the BuildBranchNodes function. This function is called once by GenerateBranches for each root node. The BuildBranchNodes function is really the heart of the tree generation process. Not surprisingly, it is a recursive function. When the initial call made from GenerateBranches returns, the entire branch node hierarchy for that root branch will have been created.

While recursive functions are often tricky to follow along with in your head, the tasks that this function must perform are really quite simple. The function is passed a branch node that has already been generated and it must create a new child node for that node. It must also decide whether the new child node (which continues the branch) should be a normal branch node or whether it should be a Branch_End node. It must also decide whether the child node it has generated will spawn additional child branches (new Branch_Begin nodes). Of course, when making these decisions, it will use the growth properties structure that the application passed in when starting the process.

When the function creates and attaches the new child node, it fills out all its information and then sends that new node into the BuildBranchNodes function with a recursive call. Therefore, each call to this function will generate a new node, attach it to the parent node (i.e., the node passed in) and recursively call itself passing in the new node as the parent node in the next recursive call. This process continues until an instance of the function determines the end of the branch as been reached and returns.

Recall from our discussion of GenerateBranches that this function was passed two parameters. The first was a root node of a root branch and the second was the current value of the BranchUID variable. The BranchUID variable was local to the GenerateBranches function and was incremented every time a root node was created. However, we want this value to be incremented for every node that is created, so we

pass this value by reference into the BuildBranchNodes call. This means that with each recursive call to BuildBranchNodes when a new branch node is created and attached to the tree, the same physical variable is accessed from each level of the recursion. The value will be increased with every node created and then assigned to that node as its unique ID.

What was not obvious in the GenerateBranches function is that BuildBranchNodes function accepts a third parameter: the current iteration count. Recall that in each node we store the current iteration which is incremented with each recursive call. This value represents the current depth of the tree where the node is stored. We use this iteration count for both determining when a branch has become too long (and should be terminated), and when calculating the texture coordinates for the branch meshes. We saw in the GenerateBranches function that each root node was assigned a value of 0. This is as we might expect because the nodes in the root level are in the first level of the hierarchy. However, that function did not pass in the iteration parameter when it called BuildBranchNodes for each root generated, which means the default iteration value of 0 will be used when adding the second node of each root branch to the tree. The result is that both the first and second nodes in a branch will have an iteration count of zero. For all nodes we add to the branch after that, it is incremented. Thus, the third branch segment will have an iteration count of 1 and the fourth will have an iteration count of 2, and so on. (it will always be incremented for all recursive calls from that point on). Is this a mistake? Not at all!

Although we have not discussed this concept previously, we do this because it is more intuitive during the tree building process if the iteration count of a branch node actually describes the index of the branch segment (cylinder) it creates. When we add the initial node, no segment is created because we need two nodes to make a segment. When the second node is added to the branch it is assigned an index count of 0 since its addition to the branch essentially adds the first segment to the branch. When we add the third node to the tree (node 2 because it is zero based), this is assigned an iteration count of 1 since its addition to the branch adds the second segment (segment 1), and so on.

Let us have a look at the code a couple of sections at a time.

```
void CTreeActor::BuildBranchNodes( BranchNode * pNode,
                                   ULONG & BranchUID,
                                   ULONG Iteration /* = 0 */ )
{
    ULONG NewNodeCount = 1, i;
    bool bEnd = false;

    // Bail if this is a branch end
    if ( pNode->Type == BRANCH_END ) return;
```

In the first section of code we set a local variable called NewNodeCount to 1. Keep in mind that, assuming the parent node we have been passed is not a branch end node, we will always add at least one node to this branch (even if it is just a Branch_End node that will cause the processing of this branch to terminate in the next recursive call). The reason we use this variable is due to the fact that we may increment this number throughout the body of the function if it is determined that the passed parent node should not only spawn another branch segment, but also one or more new branch start segments. If we decide later in the function that 2 new branches are going to be spawned from this node, this value will be set to 3. We can then create a loop that creates these three new nodes as siblings and attaches them to the parent node's (the node passed in) child list.

Notice how we initially set a boolean variable called bEnd to false. Provided the parent node is not already a branch end node, we will continue the branch by at least one more node. If at some point we determine that the child node we create should end the branch, we will set this boolean to true. This will be used to setup a branch end node instead of a normal branch node and ensure that when the next recursive call happens, and the parent branch node passed in is an end node, the recursive path will terminate all the way back to the last un-processed fork along the recursive road.

We can see this termination process being performed at the bottom of the code shown above. It tests the parent node to see if it is of type BRANCH_END. If it is, then the branch has ended and we should not add any more branch segments to that branch. In this case, we simply return. This terminates the recursive process of adding nodes to that branch.

If program flow gets past the code shown above, then it means the parent node passed into the function is not an end node and we need to continue this branch by at least one more node. What we must determine however is if this new node we are about to add to the branch is an end node. We perform a series of tests to determine this, and if any of these tests pass, the bEnd local boolean is set to true so that we know the new node we are about to create and add to this branch should be an end node.

First we test if the iteration value passed into the function (which is incremented every time the function is called and we step down another level in the hierarchy) is equal to the Max_Iteration_Count member of the tree's growth properties. This is a fixed termination level and our way of limiting the depth of the hierarchy that is ultimately created. If this is the case, then the recursive process has generated a tree that is of the maximum depth and the next branch node we add should be a branch end node. Thus, we set the branch end boolean to true, as shown below. Remember, the Iteration variable is a parameter to the function that is increased as the function repeatedly calls itself to step along the branch.

```
    // We've reached our end point here.
    if ( Iteration == m_Properties.Max_Iteration_Count ) bEnd = true;
```

Next we test to see if there is any chance of the node we are about to create spawning new child branches. In the TreeGrowthProperties structure we have several members that help us control the circumstances under which splits in the branch might happen.

First, we will definitely not want to create any splits if the iteration count passed is already larger than the maximum iteration count set for the tree. Otherwise, we would be generating a new branch at a level in the hierarchy that is deeper than we want the hierarchy to be. Therefore, we will certainly only consider splitting from this node if the current iteration count (tree depth) is smaller than the maximum iteration count set for the tree. Additionally, we only introduce new child branches at this node if the X and Y dimensions of the parent node (which are tapered as we step along the branch) are within the size ranges specified by the Min_Split_Size and Max_Split_Size tree growth properties. If either the X or Y dimensions fall outside this range, then the thickness of the branch is currently such that the application does not wish splits to happen. These properties are great for making sure that we do not spawn branches from very slim parent branches or that we do not spawn them from the base of a trunk branch where the branch is the thickest. The following code shows the conditional and the code block that will be executed only if we are allowed to spawn child branches from this node.

```
    // Chance of splitting into N ?
    if (  Iteration <  m_Properties.Max_Iteration_Count &&
          Iteration >= m_Properties.Min_Split_Iteration &&
          Iteration <= m_Properties.Max_Split_Iteration &&
          pNode->Dimensions.x >= m_Properties.Min_Split_Size &&
          pNode->Dimensions.x <= m_Properties.Max_Split_Size &&
          pNode->Dimensions.y >= m_Properties.Min_Split_Size &&
          pNode->Dimensions.y <= m_Properties.Max_Split_Size )
    {
        if ( ChanceResult( m_Properties.Two_Split_Chance ) )   NewNodeCount = 2;
        if ( ChanceResult( m_Properties.Three_Split_Chance ) ) NewNodeCount = 3;
        if ( ChanceResult( m_Properties.Four_Split_Chance ) )  NewNodeCount = 4;

        // Are we splitting here
        if ( NewNodeCount > 1 )
        {
            // Chance that a split will terminate this branch.
            if ( ChanceResult( m_Properties.Split_End_Chance ) ) bEnd = true;

        } // End if split here

    } // End if we've reached our limit
```

So what is happening inside the code block above? For starters, the fact that we are in the code block means that a new branch *can be* spawned from this node. Of course, it does not mean a new branch *will be* spawned since this will be decided using the split probability values stored in the tree growth properties structure.

Notice that we make three calls to the ChanceResult function (covered earlier). First we call it for the two split case. The chances of this test succeeding depend on the value stored in the Two_Split_Chance tree growth property. The higher the probability value we pass in, the better the odds that it will return true. We perform the two split, three split, and four split tests in that order such that higher branch splits (which typically have much lower probabilities assigned by the application) take precedence. As shown above, at the end of these tests, the NewNodeCount local variable will have been updated to contain the exact number of nodes that should be created at this level of the branch and attached to the parent node passed in as child nodes. This number is the sum of the new child branches we wish to add, plus one for the node that continues the current branch.

In the final section of the code above, you can see that provided we have introduced at least one new child branch at this node (NewNodeCount > 1), we will perform another probability test to see if the normal branch node we are about to add to the parent node (to continue the current branch) should be an end node terminating the current branch being processed. The Split_End_Chance growth probability variable is used for this test. This allows the application to control the likelihood that a branch will terminate at a node where new child branches are spawned (i.e., a fork in the branch).

At this point in the function, we know we have to create at least one new node to continue the branch, even if that node is an end node. In addition, we may also have to create one Branch_Begin nodes for each new branch starting at this new node. Forgetting about the new child branch nodes for now, let us first create a node that will be the continuation of the branch we are currently processing.

Before we create our new child branch node, we must calculate the size of this node. As discussed previously, every node in a branch will be smaller than the node before it (the parent node) so that our branches gradually taper off into tips by the time we reach the end node of that branch. In the next step we will calculate the dimensions of the new node we are about to create. How do we calculate the size of each node so that we have a gradual tapering from segment to segment such that the dimensions of the end node of the branch are zero?

We know that the dimensions of the root node describe the thickest part of the tree. That is, the X and Y dimensions of the root node describe the largest ring of vertices we will create. We also know that the TreeGrowthProperties::Max_Iteration_Count member contains the maximum depth of the hierarchy. Essentially, this tells us the iteration where the branch should have zero dimensions. Therefore, if we divide the X and Y dimensions of the root node by the maximum iteration count, we have a value that we can subtract from each node during each iteration. For example, if the root X dimension was 100 and the maximum iteration count was 25, we would calculate the amount to subtract from the dimensions of each node as follows:

SubtractAmount = 20 / 5 = 4;

So we would need to subtract 4 in each recursion from the X dimensions of the parent node when calculating the child node's dimensions. Remembering that in the next iteration, the child node will be the parent that will be reduced by another 4 units when calculating the dimensions of its child, we can see that the X dimension at each recursion would be:

Node 0 : X = 20
Node 1 : X = 16
Node 2 : X = 12
Node 3 : X = 8
Node 4 : X = 4
Node 5 : X = 0 (end node with single vertex)

As you can see, using this calculation we can recursively reduce the size of the nodes in each level of the hierarchy such that at the maximum iteration count, we have a branch tip node with zero size. We do this for both the X and Y dimensions since they may have different radii. We also do the same with the Z dimension since we want the length of each segment to get smaller as we traverse the length of the branch. However, we normally do not want the same level of downscaling from node to node as this would create very small stumpy segments at the ends of the branches. Unlike the X and Y dimensions, we are not aiming to get the length of the branch segment to be zero at the end node. To control the falloff in Z, we introduce an additional property (Length_Falloff_Scale) which can be used to allow the reduction in length from node to node to happen at a reduced rate. For example, if we set this property to 0.25, the length of the nodes will get smaller at a ratio of ¼ of the size reduction in branch thickness.

This next section of code calculates the three reduction values we need to subtract from the parent node's dimensions in order to calculate the dimensions of the new node.

```
    // Scale Ratio
    float ScalarX = (m_pHeadNode->Dimensions.x /
                    (float)m_Properties.Max_Iteration_Count);
```

```
    float ScalarY = (m_pHeadNode->Dimensions.y /
                    (float)m_Properties.Max_Iteration_Count);

    float ScalarZ = ((m_pHeadNode->Dimensions.z /
                    (float)m_Properties.Max_Iteration_Count) *
                     m_Properties.Length_Falloff_Scale );

    // Do we have enough room for another segment?
    if ( pNode->Dimensions.x < ScalarX) bEnd = true;
    if ( pNode->Dimensions.y < ScalarY) bEnd = true;
    if ( pNode->Dimensions.z < ScalarZ) bEnd = true;
```

As the above code demonstrates, after we have calculated the three reduction amounts (ScalarX, ScalarY and ScalarZ) we next test to see if any of these values are larger than the current dimensions of the parent node. If this is the case then it means the parent node is either too thin or too short to reduce by a suitable amount. When this happens, the child node we are about to add should end the branch (notice that we set the bEnd boolean to true).

At this point, we are ready to create the new child node that continues the current branch. We will deal with any new branch start nodes we may have decided to create in a moment. All we are dealing with here is the node that will continue the branch to which the parent node belongs. Here is the code that allocates a new child branch node and populates its members with the correct information.

```
    // Generate segment node (continuation of the branch)
    BranchNode * pNewNode  = new BranchNode;

    if ( !pNewNode ) return;

    // Store node details
    pNewNode->UID          = BranchUID++;
    pNewNode->Parent       = pNode;
    pNewNode->Dimensions.x = pNode->Dimensions.x - ScalarX;
    pNewNode->Dimensions.y = pNode->Dimensions.y - ScalarY;
    pNewNode->Dimensions.z = pNode->Dimensions.z - ScalarZ;
    pNewNode->Position     = pNode->Position +
                             (pNode->Direction * pNode->Dimensions.z);
    pNewNode->Direction    = pNode->Direction;
    pNewNode->Right        = pNode->Right;
    pNewNode->Type         = bEnd ? BRANCH_END : BRANCH_SEGMENT;
    pNewNode->BranchSegment= pNode->BranchSegment + 1;
    pNewNode->Iteration    = (USHORT)Iteration;
    pNode->Child           = pNewNode;

    // Clamp to minimum size, this is an end of branch
    if ( pNewNode->Dimensions.x < 0.0f ) pNewNode->Dimensions.x = 0.0f;
    if ( pNewNode->Dimensions.y < 0.0f ) pNewNode->Dimensions.y = 0.0f;
    if ( pNewNode->Dimensions.z < 0.0f ) pNewNode->Dimensions.y = 0.0f;
```

We increment the passed BranchUID member before assigning it to the node so that this node has a unique ID. We then store the address of the parent node (the node passed into this function) in the new node's Parent pointer. We also subtract the three scalar values we calculated previously from the parent node's dimensions before assigning them as the new dimensions of the child node.

Notice how we assign the new node its position. Its position is the product of adding the parent node's direction vector scaled by the parent node's Z dimensions (the length of the parent node) to the position of the parent node. Essentially, we are generating the new position by placing the child at the position of the parent, then sliding the child node along the direction vector of the parent by the Z length. This is an important concept to grasp. The Z dimension of any node actually describes the length of the segment for which that node forms its base ring of vertices. Next (in the above code) we copy the direction vector and the right vector of the parent into the child; we will deviate these in a moment.

The value we assign to the node's Type member depends on whether the local bEnd boolean was set to true in our tests. If it was, then the new node becomes a BRANCH_END type and will be the last node in that branch. Otherwise, it is assigned the BRANCH_SEGMENT type which means it is just another normal segment added to the current branch. We also store in the node's BranchSegment member the value stored in the parent node increased by one. Remember, this value will start off at zero in every BRANCH_BEGIN node so that it contains the local index of the node within the branch. The iteration value passed into the function (which would have been incremented in a previous call) is also stored in the node and then the parent node's child pointer is assigned to point at this new node. Finally, we test to see if any of the node's dimensions are smaller than zero and clamp them to zero if this is the case. This could only potentially be the case for a node that was already determined to be a branch end node.

The new node which continues the current branch has now been populated and attached to its parent, adding another segment to the branch and another level to this portion of the hierarchy. We now decide whether the child node should be randomly deviated using our ChanceResult function and the TreeGrowthProperties::Segment_Deviation_Chance member. This property holds the probability that segments within the same branch will deviate. The code that performs the deviation is shown below.

```
    // Chance of changing direction
    if ( ChanceResult( m_Properties.Segment_Deviation_Chance ) )
    {
        // Deviate the node
        DeviateNode( pNewNode,
                     m_Properties.Segment_Deviation_Min_Cone,
                     m_Properties.Segment_Deviation_Max_Cone,
                     m_Properties.Segment_Deviation_Rotate);

        // Weight with parent
        pNewNode->Direction += pNode->Direction *
                               m_Properties.SegDev_Parent_Weight;

        // Weight with growth direction
        pNewNode->Direction += m_Properties.Growth_Dir *
                               m_Properties.SegDev_GrowthDir_Weight;

        // Normalize
        D3DXVec3Normalize( &pNewNode->Direction, &pNewNode->Direction );

    } // Change direction
```

If the ChanceResult function returns true, we enter the code block that deviates the child node's direction vector. We first call the DeviateNode method passing in the minimum cone angle, the maximum cone angle, and the rotation properties for node deviation within the same branch. When this

function returns, the new node's direction vector will have been randomly deviated from its parent. However, we also have other properties to configure to restrain that deviation. For example, next we calculate how much the parent node's direction vector should be factored into the deviated vector. We add the new node's direction vector and the parent node's vector multiplied by the SegDev_Parent_Weight property (usually a value between 0.0 and 1.0). The lower this weight value, the less influence the parent node's direction vector will have over the direction vector generated for the child. Next, we add the Growth_Dir vector to the new node's direction vector. Growth_Dir is a tree growth property describing a general growth direction for the tree. The influence it will have is determined by the SegDev_GrowthDir_Weight property. After we have added these two vectors to the deviated direction vector, we renormalize it. The vector is now deviated and influenced to some degree by the parent node's vector and the general growth direction.

At this point, our child node is complete. All we have to do is continue to process the branch to which it belongs by calling the BuildBranchNodes function again in a recursive call. This time, the new child node we have just created is passed as the parent node. The current value of the BranchUID variable is also passed so that it can continue to be incremented and assigned to nodes in this branch when they are generated in future recursions. It is at this point that we also increment the current Iteration value before passing it into the next recursion. With each call into the recursive process, the Iteration value will be incremented so that as we step down the hierarchy from branch node to child branch node, the iteration values of the nodes stored at that level in the hierarchy are increased.

```
    // Generate this new branch segment
    BuildBranchNodes( pNewNode, BranchUID, Iteration + 1 );

    // Reduce NewNode Count
    NewNodeCount--;
```

One thing to bear in mind is that this function is recursively called until the parent node passed in is a branch end node, at which point it exits. When the function returns to the current iteration, all the child nodes (and any child branches spawned from those child nodes) will have been generated.

Of course, our job is not yet done for this node. We may have determined earlier in the function that additional child branches should be spawned from the node we have just created. Notice in the above code that after we have added our new node, we decrement the local NewNodeCount variable to account for the fact that we have already processed one of the new nodes we needed to add.

If the NewNodeCount value is still non-zero after the previous decrement, it means that one or more child branches (branch start nodes) will be spawned from this node. Note that these will need to be added to the node's sibling list since the new branch nodes we are about to create and the child node we just added all share the same parent.

Before we add these new branch child nodes, we must determine whether the thickness of the current branch can accommodate child branches which may be protruding at extreme angles. When we create a new node to continue a branch, we shrink its size from its parent by one step (where a step is the value contained in ScalarX, ScalarY, and ScalarZ). However, when a root node of a new child branch is about to be placed, it is likely to be deviated by a much greater angle than the deviation performed on nodes within the same branch. Since the ring of vertices at the start node of a new branch will essentially be

placed *inside* the parent branch (i.e., positioned at the same position as the child node we just generated) we want to make sure that the child branch is thin enough such that when rotated by 45 degrees for example, the vertices placed at its first node do not pierce the skin of the parent branch (Figure 12.38).

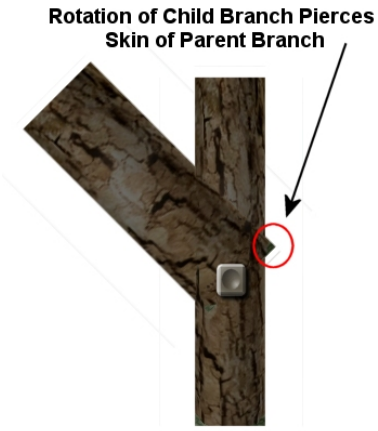

Figure 12.38

As you can see, our tree is a collection of disjoint meshes (one per branch). It is only the fact that we position the start nodes of those branches inside their parent branches that it looks like a single tree. However, as Figure 12.38 demonstrates, we need to make sure that any child branches we add are reduced enough in size with respect to the parent that we have room to rotate without the ring of vertices breaking the through the skin of the parent branch.

After some trial and error in the labs, we learned that by reducing the size of a new child branch by four steps (instead of one) we can generate children that can be freely rotated without much cause for concern. Therefore, instead of subtracting ScalarX, ScalarY, and ScalarZ from the size of the parent node as we did above when adding a node to continue the branch, when a new branch start node is spawned, we reduce its dimensions by four times that amount. So if we are at node N within a parent branch and a new branch is started at that node, the dimensions of that first node in the new branch will be equal to the thickness of node N+4 in the parent branch.

In the following code we assign this multiplier to a variable called fStepScale. We then perform tests to see if the dimensions of the parent node are smaller than the ScalarX, ScalarY, and ScalarZ values multiplied by the step scale. If the parent node is too thin to step down four levels in size, then we set the bEnd boolean to true.

```
    // Do we have enough room to split ?
    float fStepScale = 4.0f; // Reduce size by 4 times usual amount

    if ( pNode->Dimensions.x < (ScalarX * fStepScale) ) bEnd = true;
    if ( pNode->Dimensions.y < (ScalarY * fStepScale) ) bEnd = true;
    if ( pNode->Dimensions.z < (ScalarZ * fStepScale) ) bEnd = true;

    if ( bEnd ) NewNodeCount = 0;
```

If any of the dimensions of the parent node are so small that we would not be able to sufficiently reduce the size of the branch start nodes we are about to create, we set the NewNodeCount to zero. In this case, regardless of whether we determined a need to spawn child branches, no new branches will be started at this node.

If NewNodeCount is still larger than zero then we need to add one or more nodes to the parent node's child list in a loop. Each new node we are about to add in this step should be of the type BRANCH_BEGIN. For each node we allocate, we will assign it to the sibling pointer of the node that came before it in the loop, so we use the local pPrevNode pointer to store the address of the branch node that was allocated in the previous iteration of the loop. Initially we set the pPrevNode pointer to pNewNode, which is the node we allocated previously when adding another segment to the current branch. This node was assigned to the child pointer of the parent node, so any new nodes we add now should be attached to its sibling list.

```
    // Generate new split off branches ?
    BranchNode * pPrevNode = pNewNode;

    for ( i = 0; i < NewNodeCount; i++ )
    {

        BranchNode * pNewNode = new BranchNode;
        if ( !pNewNode ) continue;

        // Store node details
        pNewNode->UID          = BranchUID++;
        pNewNode->Parent       = pNode;
        pNewNode->Dimensions.x = pNode->Dimensions.x - (ScalarX * fStepScale);
        pNewNode->Dimensions.y = pNode->Dimensions.y - (ScalarY * fStepScale);
        pNewNode->Dimensions.z = pNode->Dimensions.z - (ScalarZ * fStepScale);
        pNewNode->Position     = pNode->Position +
                                 (pNode->Direction * pNode->Dimensions.z);
        pNewNode->Direction    = pNode->Direction;
        pNewNode->Right        = pNode->Right;
        pNewNode->BranchSegment= 0;
        pNewNode->Iteration    = (USHORT)Iteration;
        pNewNode->Type         = BRANCH_BEGIN;

        // Deviate the node
        DeviateNode( pNewNode,
                     m_Properties.Split_Deviation_Min_Cone,
                     m_Properties.Split_Deviation_Max_Cone,
                     m_Properties.Split_Deviation_Rotate );

        // Link the node
        pPrevNode->Sibling = pNewNode;

        // Generate this new branch
        BuildBranchNodes( pNewNode, BranchUID, Iteration + 1 );

        pPrevNode = pNewNode;

    } // Next New Branch Node
}
```

That was the final section of the BuildBranchNodes function. Notice that we assign each node a unique ID before incrementing the BranchUID variable. The dimensions we assign to each branch start node are the dimensions of the parent node reduced by four times the step reduction value used for inter-branch nodes. As before, the position of each node is simply calculated by sliding the new node from the parent node's position along the parent node's direction vector by the distance described in the parent node's Z dimension. The direction and right vectors are also copied straight from the parent node, and they will be deviated in a moment.

For each node we add in this loop we assign the BranchSegment member of that node to zero, because this member describes the index of the node relative to the start of the branch. Since we are adding branch start nodes here, the index of each new node we create will be zero. Of course, we also assign the type BRANCH_BEGIN to each node we create, as each one will begin a new branch mesh.

After we have assigned all the properties, we deviate the direction vector. We will usually want new branches to have a more obvious change in direction with respect to their parent branches so that we can see them sprouting out from the tree. When DeviateNode is called, we send in different minimum and maximum cone angles and polar rotation angles that we did before. These values specify the deviation range for nodes that start a new branch. Note that after we have deviated the direction vector of a new branch start node, we do *not* factor in the parent node's direction or the growth direction of the entire tree. The direction vector for this node will become the initial parent direction for the rest of the branch. The overall growth direction vector for the tree will be factored in eventually for the other nodes in the branch we are just starting. So even if the initial direction vector of the branch was (1,0,0), an overall growth direction of (0,1,0) would still make the other segments in the new branch start to grow upwards as they grow out.

Once the node has been created and populated, we attach it to the previous node's sibling pointer before the function calls itself recursively passing in the new node as the parent. When program flow returns back to the current instance of the function, the entire branch of nodes will have been created.

Before moving on, please make sure that you understand how the above function works by calling itself recursively to generate an entire tree of branches.

Phase One Complete

We have now covered all the code needed to implement phase one -- the building of the virtual tree. You will recall that phase one was initiated by CTreeActor::GenerateTree which was invoked by the application. The GenerateTree method called the GenerateBranches function to complete stage one. GenerateBranches called the recursive function BuildBranchNodes to build a virtual tree for each root branch node.

The GenerateTree function oversees the entire tree building process and we have only covered the first phase. Let us have another look at the GenerateTree method to remind ourselves of the functions it calls which we have yet to cover.

```
HRESULT CTreeActor::GenerateTree(   ULONG Options,
                                    LPDIRECT3DDEVICE9 pD3DDevice,
                                    const D3DXVECTOR3 & vecDimensions,
                                    const D3DXVECTOR3 & vecInitialDir ,
                                    ULONG BranchSeed /* = 0 */ )
{
    HRESULT            hRet;
    CAllocateHierarchy Allocator( this );

    // Validate parameters
    if ( !pD3DDevice ) return D3DERR_INVALIDCALL;

    // Release previous data.
    Release();

    // Store the D3D Device here
    m_pD3DDevice = pD3DDevice;
    m_pD3DDevice->AddRef();

    // Store options
    m_nOptions = Options;

    // Generate the branches
    hRet = GenerateBranches( vecDimensions, vecInitialDir, BranchSeed );
    if (FAILED(hRet)) return hRet;

    // Build the frame hierarchy
    hRet = BuildFrameHierarchy( &Allocator );
    if (FAILED(hRet)) return hRet;

    // Build the bone matrix tables for all skinned meshes stored here
    if ( m_pFrameRoot )
    {
        hRet = BuildBoneMatrixPointers( m_pFrameRoot );
        if ( FAILED(hRet) ) return hRet;

    } // End if no hierarchy

    // All is well.
    return D3D_OK;
}
```

After we have the virtual tree from phase one, it is time to move onto the phase two. Phase two is the construction of the actor's skeleton and branch skins and is initiated with a call to BuildFrameHierarchy

Notice how BuildFrameHierarchy accepts one parameter -- a pointer to an ID3DXAllocateHierachy interface. The CAllocateHierarchy class we developed has not been changed since our previous discussions. It contains both the CreateFrame and CreateMeshContainer methods that we need to allocate frames for our actor's hierarchy and convert any regular meshes we create for the branches into API compliant skinned meshes. Let us now follow the path of execution into the BuildFrameHierarchy function.

BuildFrameHierarchy – CTreeActor

The BuildFrameHierarchy function is the gateway to the recursive process for phase two of the tree generation process. It has only one task -- to call the BuildNode function with parameters that initialize the recursive process.

```
HRESULT CTreeActor::BuildFrameHierarchy( ID3DXAllocateHierarchy * pAllocate )
{
    D3DXMATRIX mtxRoot;

    // Initial frame is identity
    D3DXMatrixIdentity( &mtxRoot );

    // Build the nodes
    return BuildNode( m_pHeadNode, NULL, NULL, mtxRoot, pAllocate ) ;
}
```

The BuildNode function is passed the root node of the branch node hierarchy constructed in phase one. This function also needs to be passed a combined frame matrix that is initially set to identity (as the root node has no parent and we have not generated any frames yet). We also pass in the CAllocateHierarchy interface pointer so that the BuildNode function will be able to use its methods to generate meshes and frames. When this function returns, the entire frame hierarchy will have been constructed and populated with skinned meshes (one per branch).

BuildNode – CTreeActor

The BuildNode function is a function that recurses until the frame hierarchy and all its meshes have been created. Before we look at the code, let us briefly examine the design specifications so that we know what this function must do.

Frame Hierarchy Creation

This function will need to recursively step through the branch node hierarchy (the virtual tree) and examine each node. If a BRANCH_BEGIN node is encountered then we know that this node will be the root bone of a given mesh. Also, if we reach any type of node (except BRANCH_END) that has an iteration where Node->BranchSegment % Bone Resolution = 0, we know that we have skipped the correct number of nodes and it is time to add another bone. Remember, the bone resolution property tells this procedure how frequently it should convert branch nodes into actual bones. The BranchSegment member of a node contains its zero based index relative to the start of the branch.

Once the function determines that the current branch node is one that should be a bone, we have to allocate a new D3DXFRAME and attach it to the actor's frame hierarchy. Obviously, in the very first iteration the actor will have no hierarchy, so the first frame we create will be the root frame of the actor (the actor's m_pRootFrame member will point at this frame).

Once we have created a frame for a given node, we will also set that branch node's BoneNode boolean to true so that we will know later on that this node is a node that created a bone. This will be needed when building the skin and mesh data. We will also store a pointer to the frame we have just created in the branch node's pBone pointer. Once again, this is so we can access it later and pair branch nodes with frames.

Now that a frame has been allocated for this branch node, we need to populate it with information. Obviously, we wish this bone to be stored in the exact same position as the branch node it was created from. However, we cannot just copy over the position and orientation from the branch node into the frame's matrix. The branch node stores the position and orientation of the node as an absolute transformation from the origin of tree space, but we know that the frames must have their matrices defined in parent relative space. To address this, we will pass through the recursive process a combined frame matrix and the process will work as follows…

If we are currently adding a fifth frame to the actor's hierarchy, we will have a matrix that contains the combined transformations of the first four. We will also create a matrix for the current frame that contains the absolute position and orientation information copied over from the branch node. We now have matrix A (a combined matrix of frames 1 to 4) and matrix B (the absolute position of the current frame as taken from the branch node). We can calculate the relative matrix by inverting matrix A and multiplying with matrix B.

Relative Matrix = $B\ A^{-1}$

Essentially, we are subtracting the absolute position of the 4th frame (the parent) from the absolute position of the 5th frame (copied from the branch node) which leaves us with a matrix describing the transformation of the fifth frame relative to the fourth. Now you know how to construct relative matrices. Once we have this relative matrix, we will store it in the matrix of the frame.

Of course, we will need to pass the current combined matrix up to the point of the current branch node down to the child nodes. Therefore, using the above example, once we have calculated the relative matrix for frame 5, we will multiply that matrix with the combined matrix of frames 1-4 (passed into the function) so that we now have a combined matrix of transformations for frames 1 through 5. We will then pass that matrix into the next recursion so that it can be 'subtracted' from the absolute position of the 6th frame to generate the relative matrix with respect to its parent, frame 5. Because not all branch nodes will become bones, we need to make sure that even when we do not add a frame to the hierarchy for a node that we are processing, we still pass the combined matrix into the next recursion.

For example, imagine we have a bone resolution of 5. At the first frame, we hit a BRANCH_BEGIN node, so we add a frame (the first frame) to the actor's hierarchy. Given the bone resolution, we will not be adding another frame for this branch until we traverse another five branch nodes. However, when we hit that next branch node where a frame must be added, we will need access to the combined matrix that was generated at the last branch node a frame was created for. We need it so that we can calculate the relative matrix for the new frame. Therefore, each time the function is called, we will pass in a combined frame matrix containing

the concatenation of all the frames we have added to the hierarchy up to that point. When the branch node we are processing is one that we will create another frame in the hierarchy for, that combined matrix will be updated by combining it with the relative matrix generated for the new frame. This updated combined matrix can then be passed down to the children and used further down the branch when another frame is added. If the branch node we are currently processing is a non-bone branch node, we will simply pass along the combined matrix un-modified to the children.

Mesh Creation

The function will do more than just create frames for the actor's frame hierarchy; it will also build a CTriMesh for each branch. Once again, the action taken by the function is slightly different depending on whether we are processing a BRANCH_START node or a normal branch node.

1. If the branch node is of type BRANCH_BEGIN, we have to create a new empty mesh. We then add the initial ring of vertices to this mesh before passing this mesh down to the child branch nodes.

2. If the branch node is not of type BRANCH_BEGIN then we are currently stepping down a branch for which a mesh has already been created. All we have to do is add another ring of vertices to that mesh at the position represented by the branch node.

Every time the function calls itself for a non-BRANCH_BEGIN node, it will be passed a pointer to the mesh that was allocated when the BRANCH_BEGIN node was encountered. Therefore, we use a similar transport mechanism for the mesh as we do for the combined matrix. The branch begin node allocated the mesh, and from that point on this mesh is passed down to each child node where rings of vertices are added. By the time the BRANCH_END node has been created, we will have added a ring of vertices to the mesh for every node in that branch.

Of course, when stepping through branch child nodes and building the mesh, we may find that a node exists in a child's sibling list. This tells us that one or more BRANCH_BEGIN nodes (new child branches) start at this node. For each of these BRANCH_BEGIN nodes, we must do the same thing as before -- create a new mesh and then recursively pass it down to each of its children so that they can add their vertices. So you can imagine that as the recursive process is underway, there can be many meshes allocated but only partially built.

For example, imagine a simple case where we have a root branch with six nodes. At the first node (BRANCH_BEGIN) we create a new CTriMesh and pass it down to each of the five children in the list. By the time we have reached the branch end node, the branch will have had five rings of vertices added and one tip vertex. However, let us also assume that this root branch has a child branch that starts at node 3 (another BRANCH_BEGIN node in a sibling list with node 3 of the root branch). We would travel into that branch and process its BRANCH_BEGIN node. This would mean allocating a new mesh for this second branch and stepping through its children adding their vertex rings.

At some point we will reach the branch end node of this second branch and all vertices will have been added. The initial mesh for the root branch is still only partially complete because we took a detour after processing node 3. When the recursive process returns, we find ourselves back in the instance of the function that was originally processing node 3 of the root branch. We would now continue down to the remaining branch nodes in the root branch and complete our adding of vertices to the root branch mesh.

Figure 12.39 shows the flow of the recursive process when walking the branch node hierarchy and building and populating branch meshes.

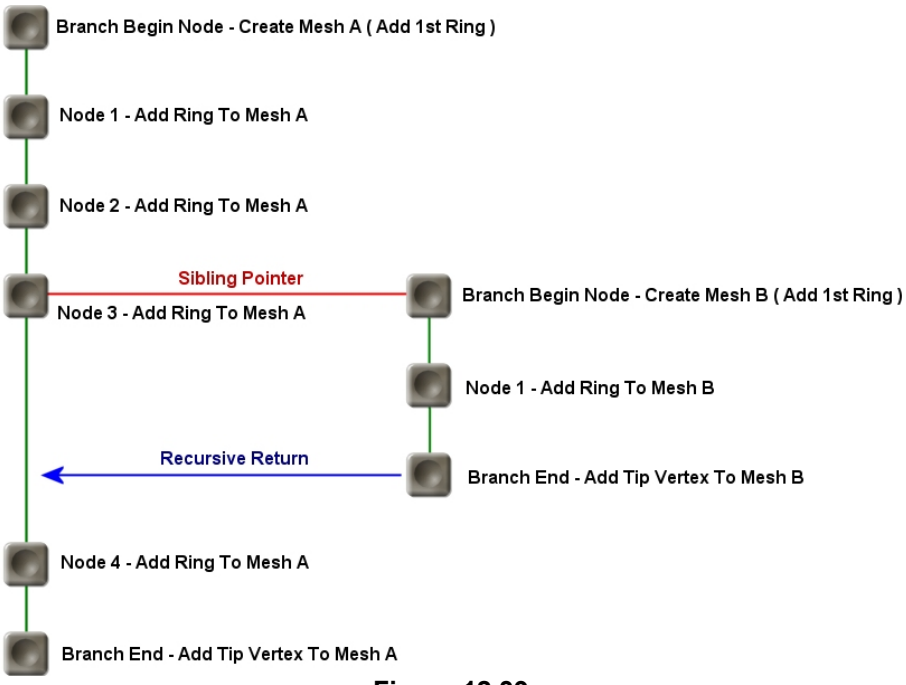

Figure 12.39

Regardless of how many branches your tree has and how many meshes it ultimately creates, the root mesh (the mesh of the trunk) will always be the first mesh allocated and the last mesh completed. We can see in Figure 12.39 that Mesh A was allocated first, but Mesh B was completed first. If you imagine more levels of recursion (child branches coming off of branch B), then you should be able to see that the last meshes to be allocated and the first to be completed are always the meshes for the smaller branches at the deepest levels of the tree.

When we add vertices to a mesh at a given node we will also add the indices to the mesh index buffer to create the branch segment (a cylinder of triangles) between the ring of vertices added at the current node and the ring of vertices added at the previous node. Remember, after the initial branch begin node, every node thereafter inserts vertices that create a new branch segment. Each ring of vertices inserted at a node (excluding the BRANCH_BEGIN node) forms the top row of vertices of the current segment being added and the bottom row of vertices for the next segment added when the child node is processed.

At the end of the BuildNode function we will once again test to see if the node we are currently processing is a BRANCH_BEGIN node. We know that if it is, then this instance of the function will have already recursively visited the child node, which would have visited its child node, and so on. Suffice to say, at this point, the recursive process has returned back to this instance of the function and the entire child branch will have had its vertices and indices added.

Recall that when we add vertices and indices to a CTriMesh, it stores them temporarily in its internal system memory arrays. Once we are sure that we have added all the vertices and index data, we call CTriMesh::BuildMesh to copy these arrays into vertex and index buffers and create the underlying D3DX mesh. Once done, we can pass the ID3DXMesh into the CAllocateHierarchy::CreateMeshContainer function to convert it into a proper skinned mesh before attaching it to the BRANCH_BEGIN frame in the hierarchy.

Of course, we must send other parameters into CreateMeshContainer, such as the material and texture lists the mesh uses. This list has only one element -- the material and the texture filename set by the application in the call to CTreeActor::SetBranchMaterial. CreateMeshContainer will take care of executing the relevant callbacks to load and store this texture and material combination.

When we covered the CAllocateHierarchy::CreateMeshContainer function, we also learned that if an intended skinned mesh is being passed in, we will also be supplied with a pointer to an ID3DXSkinInfo interface. It is this interface that provides the information needed to create the skin, such as which vertices are influenced by which bones and by what weight. It also contains the name of each bone and its accompanying bone offset matrix. For our tree, we will have to calculate all of this data ourselves, store it in an ID3DXSkinInfo object, and pass it to CreateMeshContainer. Therefore, before we call the CAllocateHierarchy::CreateMeshContainer method, we will issue a call to a helper function called CTreeActor::BuildSkinInfo to create this object for us. That allows us to treat this function like a black box until we have finished discussing the BuildNode function. We will then take a look at the BuildSkinInfo method and see exactly how we build the mapping information between vertices and bones and how we calculate the bone offset matrices for each frame used as a bone by the branch.

We are now ready to cover the CTreeActor::BuildNode function. First, let us have a look at its parameter list and describe what each parameter will be used for. This will make it much easier to then explain the code itself. The function accepts five parameters, which are listed below.

BranchNode *pNode
This is a pointer to the branch node that the instance of the BuildNode function will process. When the BuildNode function is initially called from the BuildFrameHierarchy function, a pointer to the root branch node of the virtual tree hierarchy will be passed. The function will recursively call itself using this parameter to step down to child nodes and along to sibling nodes.

It is this node that will contain the position and orientation of the ring of vertices that needs to be placed in this iteration of the function. If the node is of type BRANCH_BEGIN, the position and orientation of this node will be used to add a new frame to the actor’s frame hierarchy.

D3DXFRAME * Parent
This parameter will be NULL in the initial call. For all other recursive steps it will contain a pointer to the previous frame that was generated for the current branch (i.e., the frame that was last added to the hierarchy) and thus serves as the parent to the next frame being built along the branch. Not all nodes will create frames, so we must pass this pointer down to child nodes even if no frame was created for the current node.

CTriMesh * pMesh
This member will be set to NULL when the function is initially called from the BuildFrameHierarchy function. For all other iterations it will contain a pointer to the mesh that has been allocated for the current branch being constructed.

D3DXMATRIX &mtxCombined
This parameter will be set to an identity matrix when the function is initially called from the BuildFrameHierarchy function. For all other instances of the function, it will contain the combined (absolute) matrix of all the frames we have created so far along that path of the frame hierarchy.

When the function is processing a node that does not result in the generation of a new frame, this pointer should simply be passed unaltered into the child nodes. This allows us to continually pass the current combined matrix down through the branch node until it reaches a node where a new frame is to be generated. It is then used as part of the process to calculate that frame's relative matrix as discussed earlier in the chapter.

When the function recursively calls itself to process a sibling, it should also simply pass along the same combined matrix to the sibling that was passed into the function. The sibling frames will all share the same parent frame and therefore need to use the same parent absolute matrix to calculate their relative matrices.

ID3DXAllocateHierarchy * pAllocate
As we saw when we looked at the BuildFrameHierarchy function, this parameter always contains a pointer to an instance of our CAllocateHierarchy class. We use this object to call its CreateFrame and CreateMeshContainer callback methods whenever we wish to create a new frame in the hierarchy or build a (skinned) mesh.

With the parameter list now behind us, let us now look at the code a section at a time.

The first section of the code tests to see if the current node being processed is a node that should generate a new frame in the actor's hierarchy. The test for this is quite simple: if the node is of type BRANCH_BEGIN then this node represents the start of a new branch mesh and the initial bone of the branch. In this case, we definitely want to add a bone to the actor's hierarchy at this location. The other case in which we decide to create a new bone is if the modulus of the branch node's BranchSegment member with the tree's Bone_Resolution property is zero. Bear in mind that the BranchSegment member of a node contains the zero-based local index of the node within its branch. If a branch has 10 nodes, the BranchSegment member for each node would be in the range of 0-9, respectively. If the bone resolution was 3 for example, we would only add bones at the following nodes:

Node 1		BRANCH_BEGIN			(Bone Created)
Node 2	=	BranchSegment	Mod	Bone_Resolution	
	=	1	Mod	3 = 1	
Node 3	=	BranchSegment	Mod	Bone_Resolution	
	=	2	Mod	3 = 2	
Node 4	=	BranchSegment	Mod	Bone_Resolution	(Bone Created)
	=	3	Mod	3 = 0	
Node 5	=	BranchSegment	Mod	Bone_Resolution	
	=	4	Mod	3 = 1	
Node 6	=	BranchSegment	Mod	Bone_Resolution	
	=	5	Mod	3 = 2	
Node 7	=	BranchSegment	Mod	Bone_Resolution	(Bone Created)
	=	6	Mod	3 = 0	
Node 78	=	BranchSegment	Mod	Bone_Resolution	
	=	7	Mod	3 = 1	
Node 9	=	BranchSegment	Mod	Bone_Resolution	
	=	8	Mod	3 = 2	
Node 10	=	BranchSegment	Mod	Bone_Resolution	(Branch End = No Bone)
	=	9	Mod	3 = 0	

Remember that modulus operation returns the remainder of A divided by B as an integer. So as you can see, only when the modulus returns zero (evenly divisible) have we hit the bone resolution boundary where a new bone should be added. Of course, we always create a bone at the start of a branch. Notice that when we add the 10^{th} node, the modulus does indeed return zero, but we do not add a bone here. That is because it is the end of the branch and a bone would be unnecessary.

Let us have a look at the first section of the function and then we will discuss it.

```
HRESULT CTreeActor::BuildNode( BranchNode * pNode,
                               D3DXFRAME * pParent,
                               CTriMesh * pMesh,
                               const D3DXMATRIX & mtxCombined,
                               ID3DXAllocateHierarchy * pAllocate )
{
    HRESULT     hRet;
    CTriMesh   *pNewMesh  = NULL, *pChildMesh  = NULL;
    LPD3DXFRAME pNewFrame = NULL,  pChildFrame = NULL;
    D3DXMATRIX  mtxBranch, mtxInverse, mtxChild;
    D3DXVECTOR3 vecX, vecY, vecZ;
```

```
    TCHAR       strName[1024];

    // What type of node is this

     bool bIgnoreNodeForBone = (pNode->Type == BRANCH_END);
    if ( pNode->Type == BRANCH_BEGIN ||
        ((pNode->BranchSegment % m_Properties.Bone_Resolution) == 0 &&
         !bIgnoreNodeForBone) )
        {
        // We get here if either this is a new branch node, OR the
        // bone resolution is such that a new bone based frame is being
        // created here. Note: We don't drop here if this is an END node
        // since creating a new bone frame here would be pointless.

        // Generate frame name
        _stprintf( strName, _T("Branch_%i"), pNode->UID );

        // Allocate a new frame
        hRet = pAllocate->CreateFrame( strName, &pNewFrame );
        if ( FAILED(hRet) ) return hRet;

        // If there is a parent, store the new frame as a child.
        if ( pParent )
        {
            // Attach to head of parent's linked list
            pNewFrame->pFrameSibling  = pParent->pFrameFirstChild;
            pParent->pFrameFirstChild = pNewFrame;

        } // End if has a parent
        else
        {
            // Store at the actor's root
            pNewFrame->pFrameSibling = m_pFrameRoot;
            m_pFrameRoot             = pNewFrame;

        } // End if no parent found
```

In the above code we first determine whether a bone should be created for the current node using the techniques previously discussed. If it is determined that we should, we first build a name for that frame. Remember that in order for our frames to be animated, they must have names that will eventually match the animations inside an animation set. We need each frame in the hierarchy to have a name that is unique from any other frame, which was why we took the trouble to assign each branch node a unique numerical ID during the building of the branch node hierarchy. Thus, we build a name for the frame in the format Branch_*n* , where *n* is the ID of the branch node. We use the _stprintf (C standard runtime library) function to format the string and store the result in the local char array strName.

Now that we have the name for the frame we are about to allocate, we pass it into the CAllocateHierarchy::CreateFrame function. As we know, this is a method of our CAllocateHierarchy class that simply allocates a new D3DXFRAME derived structure, safely initializes its members and returns it. The new frame is returned to us in the pNewFrame local pointer.

Once we have our new frame, the code determines whether a parent frame currently exists (it will for all iterations of the function other than the first). If the pParent parameter is not NULL, then it contains a

pointer to the previous frame that was generated during the building process. When this is the case, we add our newly created frame to its child list. Of course, the parent frame may already have children (other branch start frames for example) so we are careful to simply add our new node to the head of the child list thus keeping the child list intact. In this case, we assign the new frame's sibling pointer to point at the current child pointer of the parent (which may point to a list of siblings), and assign the parent's child pointer to point at our new frame. In doing so, we have just added our new frame to the head of the list of child frames for the parent.

If the pParent pointer is NULL, then this is our first time through the function and the frame we have just allocated is the root frame. In this case we assign the actor's m_pFrameRoot pointer to point to our new frame, which is now the root frame for the actor's entire hierarchy. Notice however that we are still careful even in this case to attach the current value of m_pFrameRoot to the new frame's sibling pointer. Remember, there may be multiple root frames in our hierarchy that exist as siblings at the root level. Just because this frame has no parent does not mean that there is nothing useful being pointed to by the m_pFrameRoot pointer -- it may be the sibling root frame of another trunk branch -- so we should keep this list intact also.

Having created the new frame, we will also set the current branch node's BoneNode boolean to true so that we will know later that this branch node has had a bone created from it. We will also assign the branch node's pBone pointer to point at our new frame so that we have access later on.

```
        // For skin info building notify that this is the start of a new bone
        pNode->BoneNode = true;

        // Store which bone we're assigned to in the node
        pNode->pBone = pNewFrame;

        // Update the frame we will pass to the child with our new frame
        pChildFrame = pNewFrame;
```

Notice at the bottom of the above code, we assign a local pointer (pChildFrame) to the new frame we have just allocated. Because we have created a new frame, we know that it will be the parent frame of the next frame that is created further down the branch. So this pointer will serve as the frame that is passed to the child node as its parent.

We have now created a frame and attached it to the hierarchy but it currently requires initialization. Its matrix is just an identity matrix, so we need to compute the parent relative matrix. We do not know this information at the moment but we do have the absolute position of the current branch node the frame was created from, as well as the node's direction and right vectors. These vectors describe the alignment of the branch node's local Z and X axes with respect to tree space, and if we perform the cross product between the branch node's right and direction vectors we will generate the third axis of the branch node's local coordinate system.

```
        // Store / generate the vectors used to build the branch matrix
        vecX = pNode->Right;
        vecZ = pNode->Direction;
        D3DXVec3Cross( &vecY, &vecZ, &vecX );
```

At this point, we now know the orientation of the branch node's local axes as well as its position in tree space. This is all we need to build a tree space (absolute) matrix for the current frame.

```
        // Generate the frame matrix for this branch
        D3DXMatrixIdentity( &mtxBranch );
        mtxBranch._11 = vecX.x; mtxBranch._12 = vecX.y; mtxBranch._13 = vecX.z;
        mtxBranch._21 = vecY.x; mtxBranch._22 = vecY.y; mtxBranch._23 = vecY.z;
        mtxBranch._31 = vecZ.x; mtxBranch._32 = vecZ.y; mtxBranch._33 = vecZ.z;
        mtxBranch._41 = pNode->Position.x;
        mtxBranch._42 = pNode->Position.y;
        mtxBranch._43 = pNode->Position.z;
```

By placing the three axis vectors and the position into a matrix (using the specified format) we have created a single tree space matrix that describes the position and orientation of the branch node. In this coordinate system the tree root would be at the origin, but hopefully you can see that this matrix is no different from any other world matrix we would generate. Essentially, taking a point at the origin of the tree space coordinate system (0,0,0) and multiplying it with this matrix would transform the point to the position of the current node.

So, we have a matrix that describes the absolute position of our new frame, but we want it to be a parent relative matrix. We also have the combined matrix passed into the function (mtxCombined) which describes the absolute matrix of the parent frame. As discussed earlier, if we take the inverse of this combined matrix and multiply it with our current matrix, we will *essentially* subtract from our matrix all the combined transformations for all frames up to and including the parent frame. This will leave our matrix storing only the transformation 'difference' between the parent frame and the current frame's position/orientation. In other words, we have a matrix that describes our new frame as position and rotation offsets from the parent frame. In short, we have a parent relative matrix.

> **Note:** For a more technical description of what is happening here, you can refer back to Module I where we discussed the relationship between matrices, inverse matrices, and moving into and out of local coordinate systems.

```
        mtxChild = mtxBranch;

        D3DXMatrixInverse( &mtxInverse, NULL, &mtxCombined );
        D3DXMatrixMultiply( &mtxBranch, &mtxBranch, &mtxInverse );

        // Store the parent relative matrix in the frame
        pNewFrame->TransformationMatrix = mtxBranch;
```

As you can see in the above code we invert the current combined parent frame matrix and multiply it with our current absolute matrix (mtxBranch), storing the parent relative result back into mtxBranch. This is the matrix we need, so we assign our new frame's TransformationMatrix pointer to point at it. Our new frame has now been attached to the hierarchy and has received the correct matrix.

Note that the first step we took was making a copy of the absolute frame matrix (mtxBranch) to a local variable (mtxChild) before we modify it to become relative. Why? Well, think about what the mtxCombined matrix that was passed into this function contains -- the absolute matrix of the parent. When we recur down into the child node of the current node and generate a child frame, we will need

the combined matrix passed into that function to be the absolute matrix of the current frame we have just generated. Therefore, mtxChild will contain the combined matrix of the parent when processing the child node.

At this point we have created the new frame, attached it to the hierarchy, and assigned it a parent relative transformation matrix. We have also stored a local copy of the absolute transformation for our new frame so that it can be passed down to a child as the mtxCombined parameter and used to generate its relative matrix. The pChildFrame local frame pointer stores the address of our newly created frame so that it can be passed into the children of this branch as the pParent parameter. So between the mtxChild and pChildFrame local variables, we are ready for the next level of recursion into the child node.

> **Note**: At this point, please remember that we are still inside the code block that is executed only if a new frame is to be generated at this node.

In the next section of code we need to determine two things. If the current node we have just created a frame for is of type BRANCH_START then we know that this represents the start of a new branch mesh. We also need to determine what mesh we are going to pass down to the child node in the recursion. As with the mtxChild and pChildFrame local variables, we use a local variable called pChildMesh to store the mesh we need to pass down the branch.

If we are at a branch start node, we will ignore the mesh parameter passed into the function and create a new CTriMesh. We assign the pChildMesh local pointer to point at this new mesh. This will be the mesh we pass in as the mesh parameter when BuildNode calls itself for the child nodes in this new branch. If this is not a branch start node, then no new mesh needs to be created; it is just another node in a branch mesh that is already under construction. In this case, the pMesh parameter passed into the function will contain the address of the mesh that this node should pass down to its child, and the mesh it should add vertices to.

```
        // We only start creating the 'skin' mesh if this is a beginning branch
        if ( pNode->Type == BRANCH_BEGIN )
        {
            // We're going to start building a mesh, so create a new one
            pNewMesh = new CTriMesh;
            if ( !pNewMesh ) return E_OUTOFMEMORY;

            // Setup the new mesh's vertex format
            pNewMesh->SetDataFormat( VERTEX_FVF, sizeof(USHORT) );

            // Update the mesh we will pass to the child with our new mesh
            pChildMesh = pNewMesh;

        } // End if BRANCH_BEGIN
        else
        {
            // Reuse the mesh we were passed.
            pChildMesh = pMesh;

        } // End if other branch node type
```

At this point, whether we have allocated a brand new mesh, or are simply continuing the process of adding vertices to the mesh passed in for a current branch, the pChildMesh pointer will point to the mesh into which we will add the ring of vertices for this node.

Luckily for us in the short term, we use the CTreeActor::AddBranchSegment function to accomplish this task. It is passed the current node and the mesh we wish to add another ring of vertices to. We will discuss the code for the AddBranchSegment function next. Suffice to say, when it returns, a new branch segment will have been added to the branch mesh.

```
        // Add the ring for this segment in this frame's combined space
        hRet = AddBranchSegment( pNode, pChildMesh );
        if ( FAILED(hRet) ) { if (pNewMesh) delete pNewMesh; return hRet; }

    } // End if adding new frame
```

The curly brace at the bottom of the above code closes the code block that we have been examining so far. This is the code block that is only executed if the current branch node we are processing is a node for which a bone must be created and added to the hierarchy.

The next section of code is the else block of that conditional. It is executed if we are processing a node that will not add another bone to the frame hierarchy. However, we must still add a ring of vertices since every node (not just bone nodes) represents part of the branch.

```
    else
    {
        // Store which bone we're assigned to in the node
        pNode->pBone = pParent;

        // Add the ring for this segment in this frame's combined space
        hRet = AddBranchSegment( pNode, pMesh );
        if ( FAILED(hRet) ) { return hRet; }

        // Since no new frame is generated here, the child will receive
        // the same frame, mesh and matrices that we were passed.
        pChildFrame = pParent;
        pChildMesh  = pMesh;
        mtxChild    = mtxCombined;

    } // End if continuing build of previous mesh
```

As you can see, we call the AddBranchSegment function to add a ring of vertices to the mesh for this node. Notice that we pass in the pMesh parameter which contains the pointer to the mesh that has been passed into this function by the node's parent. Also notice how we set the local variables that will become the parameters for the child recursion. pChildFrame is set to the pParent frame pointer passed in since no new frame has been created and the parent frame should be passed unmodified into the child. The only time we ever wish to change the parent frame is when a new frame is inserted into the hierarchy. Finally, we set the mtxChild matrix to absolute parent matrix (mtxCombined) passed into the function. Since the parent frame is unchanged, so too is its absolute transformation. Thus, all we are doing is taking the input mesh, parent frame, and parent matrix, and preparing them to be passed down to the child.

At this point, we will now process the sibling list if a sibling exists at the current node.

```
    // Build the nodes for child & sibling
    if ( pNode->Sibling )
    {
        hRet = BuildNode( pNode->Sibling, pParent, pMesh,
                          mtxCombined, pAllocate );

        if (FAILED(hRet)) { if ( pNewMesh ) delete pNewMesh; return hRet; }

    } // End if has sibling
```

The BuildNode function calls itself recursively for the sibling node. It passes in the same parent and parent matrix as the current node (all siblings share the same parent) as well as the mesh pointer that was passed into the function.

Because of the way we organized our branch node hierarchy, the node of a branch will always be at the head of a sibling list. All other branch nodes in that sibling list (if any exist) will begin new branches. Therefore, strictly speaking, the mesh we are passing into the sibling will never be used, because as soon as we step into a sibling it will be of type BRANCH_START. This means that it will allocate its own CTriMesh for that branch and pass it down to each of its own children. However, just to safeguard against the fact that the application may have changed the order of the sibling list, we pass it the mesh. If a non-BRANCH_START node were it to exist somewhere in the sibling list, our code would still handle it correctly. The important thing to realize at this point is that when the above function call returns, all child branches that start at this node will have been *fully* built. Each sibling branch will have been traversed and created and so too will have any child branches of those child branches, and so on.

We still need to step though the remaining nodes of the current branch, so we issue the recursive call to the child pointer.

```
    if ( pNode->Child )
    {
        hRet = BuildNode( pNode->Child, pChildFrame, pChildMesh,
                          mtxChild, pAllocate );

        if (FAILED(hRet)) { if ( pNewMesh ) delete pNewMesh; return hRet; }

    } // End if has child
```

Notice this time that when we step into the child node, we pass in the pChildFrame, pChildMesh, and mtxChild local variables. What these variables store depends on whether a new frame and mesh was created at this node. Below we clarify what these variables will hold depending on the status of the current node:

Current Node = BRANCH_BEGIN node
pChildFrame = New frame added in this function
pChildMesh = New Mesh allocated in this function
mtxChild = Absolute matrix of the frame added in this function

Current Node = Normal Node (New Bone Node)
pChildFrame = New frame added in this function
pChildMesh = pMesh passed into this function from parent (pMesh)
mtxChild = Absolute matrix of the frame added in this function

Current Node = Normal Node (Non Bone Node)
pChildFrame = Parent frame passed into this function (pParent)
pChildMesh = pMesh passed into this function from parent (pMesh)
mtxChild = Absolute matrix of the frame added in this function (mtxCombined)

When the call to the child node returns, every remaining child node in the current branch will have been visited and will have added their rings of vertices to the mesh.

If the current node is not a BRANCH_BEGIN node then we have nothing left to do in this function. However, if the current node was a BRANCH_BEGIN node, then the CTriMesh for this branch will have had all the required vertices and indices added to it. It is then time to build the underlying ID3DXMesh and pass it to CAllocateHierarchy::CreateMeshContainer to have it converted to a skinned mesh. We will attach the returned mesh container to the branch start frame in the hierarchy.

```
    // If this was a 'begin' node, we can now build the mesh container
    // since all of the mesh segments should have been created by the
    // recursive calls.
    if ( pNode->Type == BRANCH_BEGIN )
    {
        D3DXMATERIAL        Material;
        D3DXMESHDATA        MeshData;
        D3DXMESHCONTAINER * pNewContainer       = NULL;
        DWORD             * pAdjacency          = NULL;
        LPD3DXBUFFER        pAdjacencyBuffer    = NULL;
        LPD3DXSKININFO      pSkinInfo           = NULL;

        // Generate mesh containers name
        _stprintf( strName, _T("Mesh_%i"), pNode->UID );

        // Generate the skin info for this branch
        hRet = BuildSkinInfo( pNode, pNewMesh &pSkinInfo );

        if ( FAILED(hRet) ) { delete pNewMesh; return hRet; }

        // Signal that CTriMesh should now build the mesh in software.
        pNewMesh->BuildMesh( D3DXMESH_MANAGED, m_pD3DDevice );
```

The first thing we do is call the CTreeActor::BuildSkinInfo function. We will look at this function shortly, but for now just know that it will create a new ID3DXSkinInfo object and will fill it with the vertex/bone mapping data. When the function returns, the pSkinInfo local variable will contain a pointer to the interface of this object. After building the skin info, we call CTriMesh::BuildMesh so that the vertices and indices we have been adding to the mesh (via the AddBranchSegment function) are used to construct the ID3DXMesh.

With our regular mesh now complete, we need to send it into the CreateMeshContainer function wrapped inside a D3DXMESHDATA structure. We set the Type member of the D3DXMESHDATA member to D3DXMESHTYPE_MESH so that the mesh container knows it is being given a regular ID3DXMesh.

```
        // Build the mesh data structure
        ZeroMemory( &MeshData, sizeof(D3DXMESHDATA) );
        MeshData.Type  = D3DXMESHTYPE_MESH;

        // Store a reference to our build mesh.
        // Note: This will call AddRef on the mesh itself.
        MeshData.pMesh = pNewMesh->GetMesh();

        // Build material data for this tree
        Material.pTextureFilename = m_strTexture;
        Material.MatD3D           = m_Material;
```

The CreateMeshContainer function also expects to be passed the texture filename and material for each subset. For our tree mesh, we have only one subset so we set up a single D3DXMATERIAL structure to store the texture filename and the material that the application set for the CTreeActor (via SetBranchMaterial).

Finally, before calling CreateMeshContainer we calculate the adjacency information for the mesh we are about to pass. We use the CTriMesh::GenerateAdjacency function, which generates the adjacency information and stores it internally inside an ID3DXBuffer object. We then call CTriMesh::GetAdjacencyBuffer which returns a pointer to its ID3DXBuffer interface and follow that with a call to ID3DXBuffer::GetBufferPointer to get a pointer to the actual adjacency information. We pass all this information into the CreateMeshContainer function which then creates a new skinned mesh (using the appropriate supported skinning method) and returns the results in the pNewContainer pointer passed in as the final parameter.

```
        // Retrieve adjacency information
        pNewMesh->GenerateAdjacency( );
        pAdjacencyBuffer = pNewMesh->GetAdjacencyBuffer();
        pAdjacency       = (DWORD*)pAdjacencyBuffer->GetBufferPointer();

        // Create the new mesh container
        hRet = pAllocate->CreateMeshContainer( strName,
                                               &MeshData,
                                               &Material,
                                               NULL,
                                               1,
                                               pAdjacency,
                                               pSkinInfo,
                                               &pNewContainer );

        // Release adjacency buffer
        pAdjacencyBuffer->Release();

        // Release the mesh we referenced
        MeshData.pMesh->Release();
```

```
        // Release the skin info
        if (pSkinInfo) pSkinInfo->Release();

        // Destroy our temporary child mesh
        delete pNewMesh;

        // If the mesh container creation failed, bail!
        if ( FAILED(hRet) ) return hRet;

        // Store the new mesh container in the frame
        pNewFrame->pMeshContainer = pNewContainer;

    } // End if beginning of branch

    // Success!!
    return D3D_OK;

}
```

When CreateMeshContainer returns, this branch will have been created as a skinned mesh and attached to a mesh container. We can then delete the original CTriMesh and the adjacency buffer before assigning the BRANCH_BEGIN frame's pMeshContainer pointer to our newly generated mesh container (which stores our branch skin).

While this was quite a complex function, remember that it is responsible for the entire hierarchy and mesh building process. When the initial instance of the function (called from BuildFrameHierarchy) returns the actor's hierarchy will be complete.

It is now time to look at the helper functions that were called from BuildNode to assist in accomplishing its required tasks. The first function we will look at is the AddBranchSegment method.

AddBranchSegment – CTreeActor

This method is passed a branch node and a pointer to a CTriMesh and it has two tasks it must perform. It must add the ring of vertices to the mesh that this node represents and it must add the indices to the mesh which connect vertices of the parent node to the vertices of the current node. This is how we form our cylinder (a branch segment) faces between the two nodes.

Things are not quite a simple as they first seem because the position of the node and the orientation of its vectors are defined in tree space. All node positions are relative to the root node of the virtual tree and therefore we might say that the node's positions and orientations are specified absolutely throughout the tree. The problem we must overcome is that each branch will be a separate mesh and we are going to want our vertex positions defined in the mesh's model space.

In the case of a branch mesh, we should consider the root node of that branch to be the origin of model space. Therefore, we want the vertex positions we add to this mesh to be defined relative to the root node of the branch, not to the root node of the entire tree. Essentially, we need to drag the root node of a

branch back to the origin of the coordinate system (dragging all its child nodes with it) so that the start node of the branch is at (0,0,0) and its local axes are aligned with the X,Y, and Z axes of the coordinate system. If we can do that, then we will have the positions and orientations of all its child nodes (in the same branch) defined in mesh/model relative space. In model space then, the start node of a branch is at the origin and all vertices are defined relative to the bottom of that branch (instead of the center of the branch which is often more typical).

Let us look at the code one section at a time.

```
HRESULT CTreeActor::AddBranchSegment( BranchNode * pNode, CTriMesh * pMesh )
{
    ULONG       i;
    USHORT      j, Index;
    D3DXMATRIX  mtxInverse, mtxRot, mtxBranch;
    D3DXVECTOR3 vecPos, vecAxis, vecRight, vecOrtho, vecNormal, vecVertexPos;
    D3DXVECTOR3 vecX, vecY, vecZ;

    // Back track until we find the beginning node for this branch
    BranchNode * pStartNode = pNode;

    while ( pStartNode->Type != BRANCH_BEGIN && pStartNode )
            pStartNode = pStartNode->Parent;

    if ( !pStartNode ) return D3DERR_INVALIDCALL;
```

In the first section of the code, we need to backtrack from the current node (which may not be a branch start node) to find the start of its branch. We will need access to this branch start node so that we can build a matrix that will transform the current node we are processing from tree space into model/branch space. As you can see in the above code, we simply start at the current node and use the pStartNode pointer to climb up through the parents of each node until we hit the start branch (a branch of type BRANCH_BEGIN). We now have the branch start node stored in pStartNode and the current node we wish to build a ring of vertices for stored in pNode.

Once we have the branch start node, we want to build a matrix for it like we did before. We perform the cross product on the right and direction vectors of the start node to get the third axis of the coordinate system. We then store the three axis vectors of the start node in a matrix along with its position.

```
    // Store / generate the vectors used to build the branch matrix
    vecX = pStartNode->Right;
    vecZ = pStartNode->Direction;
    D3DXVec3Cross( &vecY, &vecZ, &vecX );

    // Generate the frame matrix for this branch
    D3DXMatrixIdentity( &mtxBranch );
    mtxBranch._11 = vecX.x; mtxBranch._12 = vecX.y; mtxBranch._13 = vecX.z;
    mtxBranch._21 = vecY.x; mtxBranch._22 = vecY.y; mtxBranch._23 = vecY.z;
    mtxBranch._31 = vecZ.x; mtxBranch._32 = vecZ.y; mtxBranch._33 = vecZ.z;
    mtxBranch. 41 = pStartNode->Position.x;
    mtxBranch._42 = pStartNode->Position.y;
    mtxBranch._43 = pStartNode->Position.z;
```

We now have a matrix (mtxBranch) that describes the position and orientation of the local axis of the branch start node relative to the origin of tree space. For example, if we place a vertex at the origin of the coordinate system and then multiplied it by this matrix, we would transform that vertex out to the position of the branch start node.

What we want is a matrix that does the exact opposite. We want a matrix that will undo the transformation of the branch start node from any vector we multiply with it. For example, if the branch start node was positioned at 100 units along the X axis and we have a vertex positioned at 110, we want a matrix that will subtract the branch start node from the position (and orientation) of the vector. If M was our desired matrix and V was our vector, V*M should equal 10. In other words, we need to know the position of the current node we are about to process as an offset from the branch start position. Likewise, we will also want to know the orientation of the current node's direction and right vectors as rotational offsets from the branch start node's direction and right vectors. Of course we have seen this all before. What we are saying here is that we want our current node defined in the local space of the branch start node. And as we learned back in Module I (and mentioned again earlier in the lesson), multiplication by the inverse of the matrix will do the trick.

```
    // Get the inverse matrix, to bring the node back into the frame's space
    D3DXMatrixInverse( &mtxInverse, NULL, &mtxBranch );
```

Now that we have this matrix what will we use it for?

We need to create a ring of vertices and place them on a plane described by the current node. This is a plane where pNode->Direction (made unit length) would describe its normal. We will see how we place the vertices on this plane in a moment, but for now just know that we will need the direction vector, the right vector and a vector orthogonal to the them both in order to slide the vertices from the node center point out to their correct positions on the plane. However, as we wish the vertices to be placed in model space and not in tree space, it stands to reason that both the current node's position and its local axis vectors (direction and right vector) which will be used to position the vertices, should also be transformed into model space (start branch relative space) prior to being used to position those vertices

In the next section of code we transform the current node's direction and right vectors into model space using the inverse matrix. We then store them in the local variables vecAxis and vecRight, respectively. We also transform the node's position into model space and store it in the vecPos local variable.

```
    // Build the axis in the frame's space
    D3DXVec3TransformNormal( &vecAxis, &pNode->Direction, &mtxInverse );
    D3DXVec3TransformNormal( &vecRight, &pNode->Right, &mtxInverse );
    D3DXVec3TransformCoord ( &vecPos, &pNode->Position, &mtxInverse );
```

To understand what we have just done take a look at Figure 12.40. The figure shows an arbitrary branch defined somewhere in the tree. Concentrate on just the first two nodes (N and N+1) which show the first two nodes of the branch as defined in tree space (prior to the transform we have just performed). Notice the green arrows at the two nodes showing the direction vector at each node prior to transformation.

In Figure 12.40, the blue lines indicate the planes defined by the nodes in tree space. The green arrows show the direction vector of each node, which essentially describes the normal of those planes. In this example, the first node in the branch is not assumed to be the root of the entire tree but is instead assumed to be a child of some other parent branch.

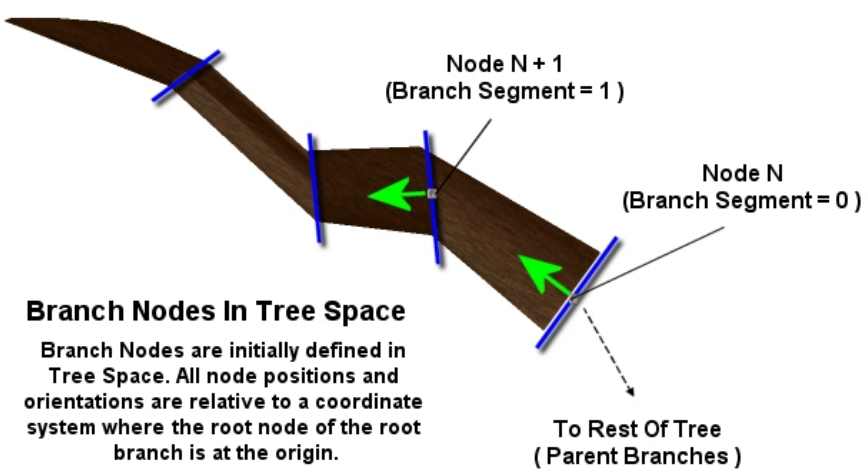

Figure 12.40 : Branch In Tree Space

In tree space, the position of each node is relative to the origin of the tree. So the position of the branch start node will not be (0,0,0) and thus is not the origin of the coordinate system. As discussed earlier, this needs to be rectified because we are going to want our children (and the vertices they represent) to be defined in the local space of the branch start node. So by applying the inverse matrix of the branch start node that we have just created to the position and orientation vectors of the current node we wish to add vertices for, we move the position and orientation of the node into the coordinate space defined by the branch start node. In Figure 12.41 you can see how all the nodes for the branch shown above look once transformed into mesh local (i.e., model) space.

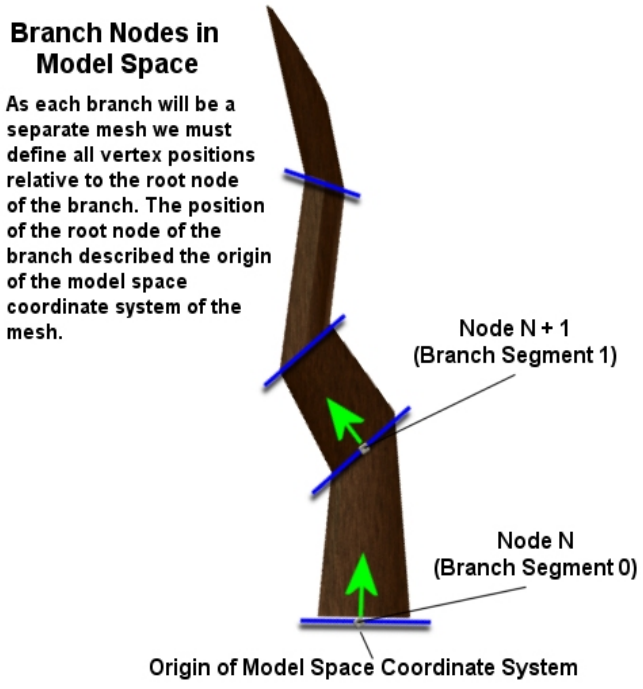

Figure 12.41 : Branch Nodes in Model Space

As you can see, in model space the nodes of the branch represent a unique mesh. The direction and right vectors of the branch start node form the Z and X axes of the coordinate system and the tree space position of the branch start node is mapped to the origin of the coordinate system.

Notice how the orientations and direction vectors in each node have changed. Yet while very different from their tree space counterparts, they still maintain the same inter-branch relationship.

It is the direction and right vector of the current node in this space which should be used to position the ring of vertices on the model space plane defined by each node. After all, when we created the start node for this branch we would have also added a frame to the actor's hierarchy. Remember, it is the frame matrices in the hierarchy that are responsible for ultimately transforming this model space branch into its proper tree space position at render time.

At this point, we now have the direction vector and the right vector of the current node in model space. The direction vector describes the normal of the plane on which the vertices should be added and the right vector is tangent to the plane (it lies on the plane). In order to position our vertices on this plane we will also need an additional tangent vector (often referred to as a binormal). If you imagine you are staring at a clock (with hands) that is currently showing 3 o'clock, think of the direction vector as an arrow coming out of the center of the clock pointing right at you. Now imagine that the little hand that is on the 3 is the right vector of the node. Finally, the binormal would be the big hand of the clock pointing at the 12 in this particular case.

If we have two tangent vectors, we have the ability to place vertices anywhere on that plane by combining those vectors and scaling. This should not surprise you since we are basically talking about a standard Cartesian XY plane here (where the direction vector is the Z axis). The X axis (1,0,0) and a Y axis (0,1,0) are both tangent vectors for the XY plane and they allow us to position vertices anywhere on that plane. If we want to place a point at position (40, 50) on the XY plane, we think of this as describing a point 40 units along the X axis and 50 units along the Y axis. However, what we are really saying is this:

Position X = 40 * (1, 0, 0) = (40, 0, 0)
+
Position Y = 50 * (0, 1, 0) = (0, 50, 0)
=
Position XY = (40, 50, 0)

This is a more mathematically correct way of thinking about what happens when we plot a coordinate. The reason we get the expected results is that our transformation matrices store an X axis and a Y axis with a length of 1.0 (unit length). But if our transformation matrix stored X and Y axis vectors with a length of 3.0 instead, the final position plotted on that plane is obviously entirely different:

Position X = 40 * (3, 0 ,0) = (120 , 0 , 0)
+
Position Y = 50 * (0, 3, 0) = (0, 150, 0)
=
Position XY = (120, 150, 0)

This is the mathematics of linear transformations and vector spaces. We do not need to get much more technical here in this course since we expect that you will cover such material in the Game Mathematics course. If you have not done so already, you should get used to viewing transformations in this way.

The point of this exercise was to demonstrate that if we have two unit length tangent vectors, we can use them to plot any point on the plane they define. When we position our vertices on the plane of the node, we will not be using the usual X and Y axis vectors as shown above. The plane of the node may be rotated and/or tilted in model space, so we will use the tangent vectors instead.

At the moment we currently have one tangent vector -- the node's right vector. We can think of this as the node's local X axis. We need a local Y axis also so that we can use these two vectors to position the ring of vertices on the plane.

Calculating this vector is easy and you should already know how to do it. The right vector is tangent to the plane and that the direction vector is perpendicular to the plane. The other tangent vector we wish to create should be orthogonal to these two, so all we have to do is rotate the model space right vector around the model space direction vector by 90 degrees and we have our second tangent vector (the binormal). This is shown in Figure 12.42.

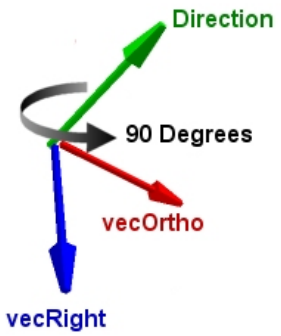

Figure 12.42

So let us build our node's third model space axis next:

```
    // Build the ortho vector which we use for our Y dimension axis
    D3DXMatrixRotationAxis( &mtxRot, &vecAxis, D3DXToRadian( 90.0f ) );
    D3DXVec3TransformNormal( &vecOrtho, &vecRight, &mtxRot );
```

We now have the node position and its local axis in model space so it is time to start adding our vertices. The second section of the function is divided into two code blocks. The first is executed only if the current node is not of type BRANCH_END. It adds a ring of vertices on the model space node plane and adds the indices to the mesh to stitch them into faces with the previous node's ring of vertices. The second code block is executed only if we are currently processing the BRANCH_END node. When this is the case, only a single vertex is added at the node position (not a ring) and the indices are added to stitch this final vertex into faces using the previous node's ring of vertices. The end result is the branch tip.

Let us first have a look at the code block that is executed for nodes that are not BRANCH_END nodes (adding rings).

```
    // If this is a beginning / segment node
    if ( pNode->Type != BRANCH_END )
    {
        // Add enough vertices for the new branch segment
        long VIndex = pMesh->AddVertex( m_Properties.Branch_Resolution );

        // Generate the vertices
        for ( i = 0; i < m_Properties.Branch_Resolution; ++i )
        {
            CVertex * pVertex = &((CVertex*)pMesh->GetVertices())[ VIndex + i ];

            // Calculate angle of rotation ((Branch_Resolution - 1)
            // because we are duplicating one vertex at the seam)
```

```
            float fAngle =
            D3DXToRadian( (360.0f / (float)(m_Properties.Branch_Resolution - 1))
                          * -(float)i );

            // Plot the points for our 'elliptical' branch hull
            float x = pNode->Dimensions.x * cosf( fAngle );
            float y = pNode->Dimensions.y * sinf( fAngle );
```

In the above code, we first tell the mesh that we are about to add a ring of vertices so that it can make room for those vertices at the end of its internal vertex arrays. We do this using the CTriMesh::AddVertex method and pass the number of new vertices we wish to make space for. The number of vertices that will be used to form a ring is stored in the Branch_Resolution member of the growth properties structure (8 by default). The AddVertex method returns the index (VIndex) of the first vertex in the array of vertices we have just added.

We then entered a loop which iterates through each vertex we wish to add and allows us to calculate the position of each new vertex in the ring one at a time. Inside the loop you can see that we retrieve a pointer to the vertex structure whose position and normal we will need to set. Notice how we use the base index of the first new vertex we allocated (VIndex) and add that onto that loop variable 'i' to allow us to step through each of the new vertices.

The next thing we did was calculate a delta angle (fAngle). If we have 8 vertices to place (e.g., Branch_Resolution = 8), then we need to step around the model space position of the node and position vertices in a circle at uniform intervals. As the first and last vertices have duplicated positions for the wrap around, we need to step around a 360 degree circle in increments of (360 / Branch_Resolution - 1). Using the default branch resolution of 8, this means for each iteration of the loop, we will need to step around the circle surrounding the node a further 360/7 = 51.42 degrees and place another vertex.

As 51.42 degrees describes the size of single wedge formed by two adjacent vertices in the ring (with the center point as the apex of the wedge), we can multiply this value with the current loop iteration variable 'i' so we know exactly where we need to place the vertex. For example, in the first three iterations of the loop, fAngle would equal the following:

Iteration 1 : 51.42*0 = 0 (Vertex placed at zero degrees about circle)
Iteration 2 : 51.42*1 = 51.42 (Vertex placed at 51.42 degrees about circle)
Iteration 3 : 51.42*2 = 102.84 (Vertex placed at 102.84 degrees about circle)

Once we have the rotation angle for the current vertex, we can calculate a position on the circumference of the ellipse with the help of the sine and cosine functions. As we know, the sine and cosine functions are 90 degrees apart, so if we wish to place a point on a circle with a radius R at rotation angle A, we calculate the X and Y coordinates of that point as:

X Position = R * cosine (A)
+
Y Position = R * sine (A)
=
(X Position , Y Position , 0)

Notice that we are ignoring the Z coordinate for the moment since we are essentially defining a 2D circle on a plane. At this particular point in time, that plane is actually the XY plane of the model space coordinate system. The circle will also be defined about the origin of the coordinate system at this point.

In our code however, the node's Dimensions.X and Dimenions.Y members define the radii of the ellipse. (If these two members are equal then we define a circle.), Since the X radius may be different from the Y radius, this turns our calculation into:

X Position = pNode->Dimensions.x * cosine (A)
+
Y Position = pNode->Dimensions.y * sine (A)
=
(X Position , Y Position , 0)

Since the cosine and sine are 90 degrees apart, they allow us to apply scaling factors to the X and Y dimensions such that the final result will always lie on the circumference of the ellipse. For example, if the angle is 0 degrees (first iteration) then the cosine will return 0 and the sign will return 1. This will result in a first vertex position of:

(pNode->Dimensions.x , 0 , 0)

The start of the circle is not the top but actually the position of the number 3 in the clock face example (along the X axis). However, at 90 degrees, the cosine returns 0 and the sine returns 1, which results in a final vector at the top of the circle:

(0 , pNode->Dimensions.y , 0)

At 45 degrees the angle is equidistant from both the X and Y axis and the result of both will be ~0.707106. In this instance we are scaling both the X and Y radii of the ellipse to produce a point that is between both axes, but still the correct distance from the circle center point (on the circumference of the circle).

If we imagine that we are calculating the position of a vertex at 45 degrees around the circle of a node that has both an x and y dimension of 1, this will result in the following X and Y coordinates:

X Position = 1 * 0.707106
+
Y Position = 1 * 0.707106
=
(0.707106 , 0.707106 , 0)

As you can see, this is exactly the same vector you would get if you normalized the vector (1, 1, 0), which we know would result in a 45 degree vector describing a position that is halfway between the X and Y axes but still has a length of one.

At the moment we are simply defining the positions of each vertex as if it was being projected onto the XY plane of the model space coordinate system. The center of the circle would be the coordinate system origin. Although we process each vertex one at a time, if we were to imagine the positions of every vertex in the ring at this point in the loop, we would see a result like Figure 12.43.

As Figure 12.43 shows, the dimensions of the current node define the radius of the ring of vertices around its center point. However, at the moment, the vertex positions are not defined relative to the node's plane. That is to say, the center of this circle is currently (0,0,0) and not the node's model space position. Furthermore, the vertices currently lay flat on the XY plane of the coordinate system and not the plane of the current node.

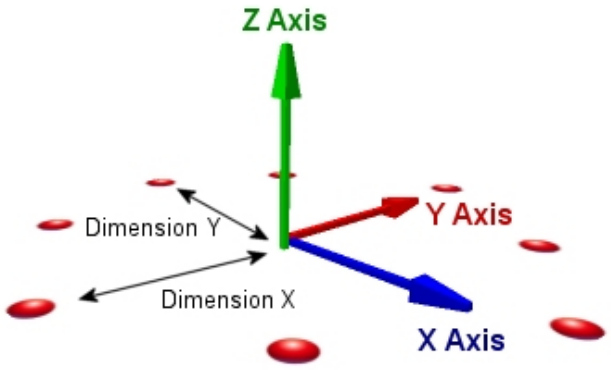

Figure 12.43 : Circle of Vertices in model space

The first thing we must do now that we have our vertex position defined with respect to the center of the ellipse is position that vertex on the node plane. The thing to remember is that the node plane may be oriented quite differently from the XY plane of the model space coordinate system. It will most likely be rotated and/or tilted in some way. So given a vertex position on the XY plane of the model space coordinate system, how do we place it in its corresponding position on another plane with a totally different orientation?

A little while ago, we said that when we specify a coordinate in a Cartesian coordinate system, we are really stating that these components should be multiplied with the X, Y, and Z axes of the coordinate system in which we are trying to plot them. Because we usually mean to plot such points in a coordinate system with the axes (1,0,0), (0,1,0) and (0,0,1), the resulting multiplication with each axis does not alter the input position of the vector. Therefore, while there was no need to perform the calculation explicitly, mathematically speaking, we calculated the position of our vertex as follows:

X Position = pNode->Dimensions.x * cosine (A)) * (1,0,0)
+
Y Position = pNode->Dimensions.y * sine (A)) * (0,1,0)
=
(X Position , Y Position , 0)

Of course, the result is the same, so there was no need to supply the additional model space axis multiplications on the end. However, Figure 12.45 shows how we might want the circle of vertices to look on the actual node plane. Remember it may be tilted or rotated with respect to the XY plane of the coordinate system shown in Figure 12.44.

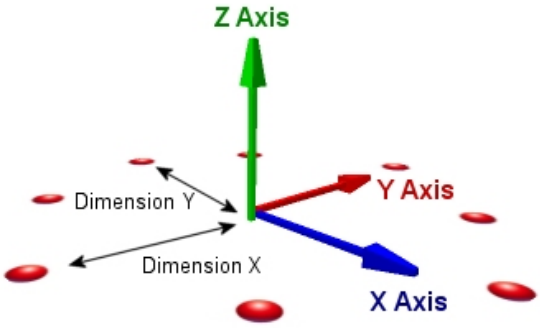

Figure 12.44 : Vertices multiplied by Model Space X and Y axis

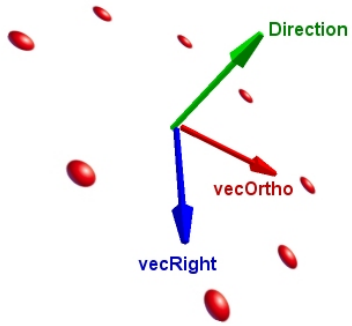

Figure 12.45 : Vertices multiplied by node tangent vectors

So let us piece together some of what we already know. When we positioned the vertices on the XY plane of the coordinate system, we were actually multiplying the X and Y position with the X and Y axes of the coordinate system shown in figure 12.44. We did not have to perform this calculation explicitly and the lack of such a calculation means that it was implied.

In Figure 12.45 we see that our node plane also has its own X and Y axes which are tangent to the plane (vecRight and vecOrtho). Therefore, all we have to do is multiply the vertex position that is currently on the model space XY plane with the node's local axes and we can move each vertex onto the plane.

```
            // Set and scale the vertex position based on the chosen dimension.
            vecVertexPos = (vecRight * x) + (vecOrtho * y);

            // Generate our normal from this position
            D3DXVec3Normalize( &vecNormal, &vecVertexPos );
```

As you can see, we multiply the node's right vector (X axis) with the vertex X position and the node's other tangent vector (the Y axis) with the vertex Y position and then sum the resulting vectors.

At this point, the plane the vertex is on is not really the node plane; it is just has the same orientation as the node plane. It still passes through the origin of the coordinate system (it has a distance of zero). In a moment we can fix this by simply adding the model space position of the node to the vertex position to move it into place on the node plane.

But before we do this, notice that in the above code we calculate a normal for the vertex simply by taking the position of the vertex and normalizing it. Remember, at this point, although we have oriented the plane (and its vertices) the origin of the coordinate system is still the center of the ellipse. Therefore, the vertex position itself represents a vector from the center of that circle out to the vertex.

Figure 12.46 illustrates this concept for a single vertex defined on the oriented plane with the origin of the coordinate system still at the center of the circle.

The black arrow pointing from the center of the circle is the vector that is represented by the vertex position at this time. If we normalize this vector, it actually serves as a fairly good vertex normal for our purposes. After the mesh is complete, if you prefer, you could do an averaging sweep through the mesh such that the vertex normal is actually the average of both the face normals to the left and right of it. Actually, most vertices will belong to four faces as it will form the top row of vertices for one cylinder and the bottom row of vertices for the next cylinder. However, we do not perform this averaging step and simply use this normal 'as is' which seems to give fairly good results when lighting the tree.

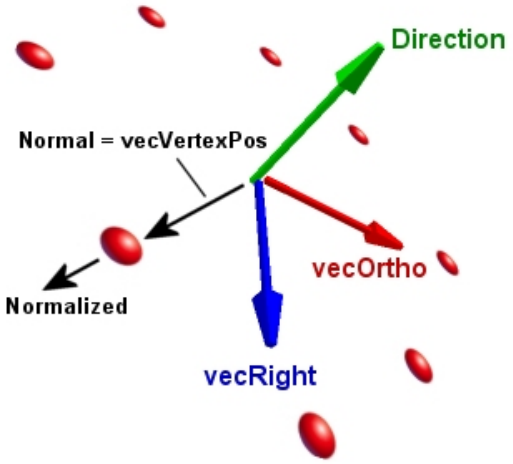

Figure 12.46 :
Normalized Vertex Pos = Vertex Normal

As mentioned, the vertex is oriented on the plane correctly but the center of the circle is still the origin of the coordinate system, not the node position. That is easily fixed by adding our current vertex position and the position of the node and storing the result (along with the normal) in our vertex structure.

```
            // Push out to it's final position, relative to the branch node
            vecVertexPos += vecPos;

            // Store the values in the vertex
            pVertex->x = vecVertexPos.x;
            pVertex->y = vecVertexPos.y;
            pVertex->z = vecVertexPos.z;

            // Store the vertex normal
            pVertex->Normal = vecNormal;
```

Before finishing this vertex we have to assign it a pair of UV texture coordinates. We discussed the technique used for UV calculation earlier in this lesson. The U coordinate is assigned a position across the width of the texture which is a product of the vertex index 'i' within the ring divided by the total number of vertices comprising a ring. This assures that vertex[0] in the ring is mapped to a U coordinate of [0] and vertex [N] is mapped to a U coordinate of 1.0 (where N is the branch resolution). All other vertices between the start and end vertices of the ring will be uniformly mapped across the width of the texture. We also scale the final result by the Texture_Scale_U growth property to allow tiling and/or texture sizing. If left at 1.0, the width of the texture will be wrapped around the cylinder/branch segment once.

Notice that we use Branch_Resolution - 1 because the first and last vertices are in the same position. We want the first vertex to have a U coordinate of 0.0 and the last vertex to have a U coordinate of 1.0 even though they are at the same position in the ring. If we do not, our mapping will be incorrect in the area between the last and first vertex. For example, if we had a ring of 10 vertices where each had its own unique position in the ring (no duplicates), the U coordinates assigned to the last two vertices would be

0.9 and 1.0. We can see that when mapping the face between the last two vertices, the texture would have its final 1/10th (the interval 0.9 to 1.0) copied between vertices 9 and 10 in the ring. This is absolutely correct. However, when the texture is mapped to the final face in the ring between vertex 10 and vertex 1 (to wrap around) we would be mapping between a vertex with a texture coordinate of 1.0 and the first vertex with a texture coordinate of 0.0 which gives us a mapping interval of 1.0 - 0.0. In other words, the entire width of the texture (in reverse) would be mapped into the space between the last and first vertex.

By duplicating the first and last vertex positions and assigning them different texture coordinates, we can rest assured that we have two U texture coordinates at that initial vertex position in the ring. For the first two vertices in the ring in our current example the U interval would be 0.0 – 0.1 and for the last the two vertices the interval would be 0.9 - 1.0 (instead of 1.0 - 0.0 which would otherwise be the case if the first vertex was used to complete the last face).

```
            // Generate texture coordinates
            pVertex->tu = ((float)i / (float)(m_Properties.Branch_Resolution - 1))
                            * m_Properties.Texture_Scale_U;

            if ( !pNode->Parent )
                pVertex->tv = 0.0f;
            else
                pVertex->tv = ((float)(pNode->Iteration + 1) /
                               (float)m_Properties.Max_Iteration_Count)
                                    * m_Properties.Texture_Scale_V;

        } // Next Vertex
```

Notice when calculating the V coordinate, if the current node has no parent then it is simply set to 0 (the bottom/top of the texture). Remember from our earlier discussion that unlike the U coordinate, the V coordinate is not local to the ring. In fact, when the Texture_Scale_V growth property is at its default value of 1.0, the height of the texture is mapped to the entire height of the tree. Thus, its level in the virtual tree hierarchy determines the node's V texture coordinate. This is just a simple case of dividing the node's iteration (depth in the hierarchy) by the maximum iteration (maximum depth of the hierarchy) to map its iteration into the 0.0 to 1.0 range. Notice that we actually use Iteration + 1. This is because (as discussed earlier) the first two levels of the branch node hierarchy have the same iteration of zero. As the first iteration will have no parent and will always be assigned a V value of 0 in the above code, we want the second node to be 1 not 0, and the third node to be 2 not 1, and so on.

And there we have it. We have now positioned all the vertices in the ring for this node, assigned them normals and texture coordinates, and added this information to the CTriMesh.

Unfortunately, our work is still not done. We still have to add indices to the mesh also so that we form a band of triangles that join all the vertices in the ring we have just added with the vertices added by the parent node (in a previous AddSegment call). Of course, we only perform this step if the current node is not a BRANCH_BEGIN node, as this would mean we would have only added one ring/row of vertices. We need at least two rings of vertices to create a branch segment, so the following code will only be executed for the second node and beyond in a given branch. Therefore, if the current node is a BRANCH_BEGIN node, we would have nothing left to do in this function.

Let us have a look at the code block that starts to add the indices to the mesh.

```
        // If this is not the start of a new branch, add indices
        if ( pNode->Type != BRANCH_BEGIN )
        {
            ULONG FaceIndex=pMesh->AddFace((m_Properties.Branch_Resolution-1)*2);
```

The first thing we do is call the CTriMesh::AddFace method to reserve space to add indices to the mesh's indices array. Two rings of N vertices allow us to create N-1 quads because there are N-1 pairs of vertices in an N vertex ring with which to form the base of a quad. We also know that if we have two rows of vertices with 8 vertices in each, then by stepping around the ring of vertices a vertex at a time and using two vertices from each ring, we could construct 7 quads to form the hull of the cylinder segment. Furthermore, as each quad would need to be constructed from 2 triangles, we know that with a default branch resolution of 8 we would need to create 7*2=14 triangles to wrap a ring of faces around our rings of vertices and form another branch segment.

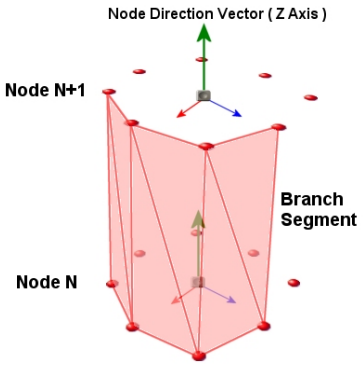

Figure 12.47

In Figure 12.47 we see two rings of vertices with a branch resolution of 8 (which actually creates 8 unique vertex positions that can be seen). While we have not filled in all the faces, you should be able to fill in the blanks yourself and see that in this case we would need to allocate space for 8*2 = 16 triangles. Node N+1 is actually the current node that we are processing and Node N is the ring of vertices that was added to the mesh when the previous/parent node was processed.

Remember that the CTriMesh::AddFace method returns the index of the first face in the array of new triangles that have been added to the mesh. We can use the CTriMesh::GetFaces method to get a pointer to the indices array and offset a pointer to the first new index we just allocated and now must populate. For example, let us assume we have just added 16 faces to the mesh (via the above call) and it returned us a face index of 80. This means that face 80 in the mesh is the first face of the 16 that we just allocated in the face array (80 have been added in previous function calls). Therefore, faces 80 through 95 will be the faces that represent the segment we are about to add.

Although we can use the CTriMesh::GetFaces method to fetch a pointer to the indices array, we only want to alter the indices used by faces 80 through 95. That is no problem since we already know the index of the first new face we added. Since all faces are triangles with three vertices, we just have to make our index pointer offset (80 * 3) into the array in this example. That is exactly why CTriMesh::GetFaces returns that useful 'first new face' index.

```
            // Retrieve the face index array
            USHORT * pIndices = &((USHORT*)pMesh->GetFaces())[FaceIndex * 3];

            USHORT Row1 = (pNode->Parent) ? pNode->Parent->VertexStart : 0;
            USHORT Row2 = (USHORT)VIndex;

            // Store this nodes vertex start
            pNode->VertexStart = Row2
```

What are the Row1 and Row2 locals doing in the above code? We need to access two rings of vertices in order to build this new branch segment. We need to know the indices for the vertices we have just added for the current node's ring as well as the indices of the parent node's ring so that we can stitch them together to make faces. Earlier in the function when we called CTriMesh::AddVertex to make room for our ring of vertices at the current node we were returned VIndex. VIndex describes the start location (first vertex) in our mesh's vertex array for the ring of vertices we have just added. This will be the second row of vertices that will form the top of the branch segment we are about to create. Notice in the above code how the node also stores the index of the first vertex used by this ring. When another branch segment is added to this branch, when we visit the child node, we can simply fetch this value from the parent node to know where its vertices start. You see this happening in the calculation of Row1 in the above code.

If the current node we are processing has no parent, then this is the first segment we have added (second node of the entire tree) and we know that the vertices for the previous node (the branch begin) must start at zero. If it does have a parent node then we fetch the index at which its vertices start in the current mesh (via its VertexStart member). As this code is only executed when we are not processing a branch start node, the VertexStart member for a branch start node will be left at its default value of zero. This is as it should be, because for any individual branch, its first ring of vertices will be created from the branch start node and will be at the very beginning of its vertex buffer.

At this point, we have an index telling us where the parent node's ring of vertices start in the vertex array and an index telling us where the current node's vertices have been added in that same array. All that is left to do now is loop around the ring of vertices and add the indices of the triangle. With each iteration of this loop we will add a quad to the index buffer (i.e., two triangles, 6 indices). Since pIndices currently points to the position in the mesh's index array where we want to start adding this index data, we can simply increment a counter variable (Index) by 6 during each iteration of the loop.

Hopefully, the following code will make sense to you (you may need to review it a few times and work through it on paper). For the first triangle of the quad we index two vertices from the previous node's ring and one vertex from the current node's ring. For the second triangle of the quad we use two vertices from current ring of vertices and one from the previous node's ring.

```
            // For each new face
            for(j=0, Index=0; j<m_Properties.Branch_Resolution-1;++j, Index+= 6 )
            {
                // Add the indices for the first triangle
                pIndices[ Index     ] = Row1 + j;
                pIndices[ Index + 1 ] = Row2 + j;
                pIndices[ Index + 2 ] = Row1 + j + 1;

                // Add the indices for the second triangle
                pIndices[ Index + 3 ] = Row2 + j;
                pIndices[ Index + 4 ] = Row2 + j + 1;
                pIndices[ Index + 5 ] = Row1 + j + 1;

            } // Next Face
        } // End if not beginning of branch
    } // End if not end node
```

As you can see, we are adding six indices (two triangles) with each iteration of the loop and using loop variable 'j' to step through the vertices in the top and bottom rows each time.

We have now seen all the code that is executed to add a segment when the current node is not a node of type BRANCH_END. When the current node is an end node things have to be done a little differently. We no longer add a ring of vertices at the current node; instead, we just add one vertex at the model space node position. This vertex is essentially being positioned at the center of the ring of vertices that would have been generated for this node were it not an end node.

```
    else
    {
        // Add just the one tip vertex.
        long VIndex = pMesh->AddVertex( 1 );

        CVertex * pVertex = &((CVertex*)pMesh->GetVertices())[ VIndex ];

        // Same as the node position
        vecVertexPos = vecPos;

        // Store the values in the vertex
        pVertex->x = vecVertexPos.x;
        pVertex->y = vecVertexPos.y;
        pVertex->z = vecVertexPos.z;

        // Store the vertex normal
        pVertex->Normal = vecAxis;
```

The normal calculation is also different for this vertex since we can no longer create a normal using a vector from the center of the circle to the vertex. This is obviously because the vertex is at the center of this circle. Instead, we just use the model space direction vector of the end node as calculated in the virtual tree generation process.

Generating texture coordinates for this vertex is simple also. The U coordinate is always set to 0.5 so that it will be mapped to a point exactly half way across the width of the texture (if no scaling is being used). This seems logical if you consider that this is a tip in the center of where the circle would be, and as such is halfway between both sides of that circle. The V texture coordinate is calculated the same way as before (i.e., a function of node hierarchy depth).

```
        // Generate texture coordinates
        pVertex->tu  = 0.5f;
        pVertex->tv  = ((float)(pNode->Iteration + 1) /
                        (float)m_Properties.Max_Iteration_Count)  *
                         m_Properties.Texture_Scale_V;
```

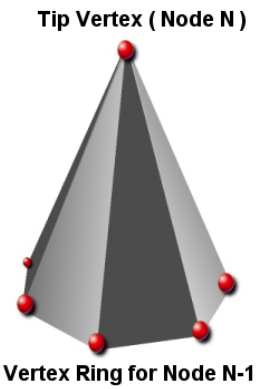

Figure 12.48

Now it is time to add the faces for this end segment. Although we added only one vertex, we still need to form a triangle between this vertex and every vertex in the ring inserted at the previous branch node. This allows us to end the branch using a cone shape (Figure 12.48).

If we have a ring of N vertices at the previous node then we know that these can be used to build the base of N-1 faces. We can see that the number of faces we wish to add to the CTriMesh in order to add this cone would be the branch resolution minus 1. As before, we will use the CTriMesh::AddFace function to allocate this many faces (essentially just multiplies the desired face count by three and allocates that many indices in the mesh's indices array).

```
        // Creating pointed tip
        ULONG FaceIndex = pMesh->AddFace( m_Properties.Branch_Resolution - 1 );

        // Retrieve the face index buffer
        USHORT * pIndices = &((USHORT*)pMesh->GetFaces())[FaceIndex * 3];

        USHORT  Row1 = (pNode->Parent) ? pNode->Parent->VertexStart : 0;
        USHORT Row2 = (USHORT)VIndex;

        // Store the vertex start
        pNode->VertexStart = Row2;
```

Notice that once again, after allocating space for the new indices, we use the CTriMesh::GetFaces function to return a pointer to the start of the index array. We then multiply the index of the first new face we have allocated by 3 to offset the indices pointer so that it points to the first new index position we must fill in.

In the following and final section of code for this function, we loop around each vertex in the ring forming a triangle with the tip vertex (Row2) and each pair of vertices from the previous ring (Row1 and Row1+1).

```
        // For each new face
        for (j=0, Index=0; j<m_Properties.Branch_Resolution - 1; ++j, Index+= 3 )
        {
            // Add the indices for the first triangle
            pIndices[ Index     ] = Row1 + j;
            pIndices[ Index + 1 ] = Row2;
            pIndices[ Index + 2 ] = Row1 + j + 1;

        } // Next Face

    } // End if end node

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

We have now covered the complete AddBranchSegment function. Remember that it was called with each iteration of the BuildNode function to add a ring of vertices and indices at the current node being processed. Although on first read the function may seem quite intimidating, you should notice after further study that it really is rather intuitive (although admittedly a little long).

There is one more function we must cover before we have essentially covered the entire tree generation process. You will recall from our coverage of the BuildNode method that the BuildSkinInfo method was called whenever we were processing a branch start node (BRANCH_BEGIN). Let us have a look at this function next.

BuildSkinInfo - CTreeActor

The CTreeActor::BuildSkinInfo function is called towards the end of the BuildNode function only if the current node being processed is of type BRANCH_BEGIN. At this point in the BuildNode function, the child nodes of the BRANCH_BEGIN node will have already been visited and all the vertices and indices of each node in this branch added to the new CTriMesh.

Recall that after our CTriMesh has been built, we pass it into the CAllocateHierarchy::CreateMeshContainer function to convert it into a skinned mesh. However, this function requires that we send it an ID3DXSkinInfo object containing the information about each bone in the hierarchy that influences this mesh and the vertices in the mesh that they influence. Traditionally, we have had the ID3DXSkinInfo created for us by D3DX when loading skinned meshes from X files, but because we have hand-crafted this branch mesh ourselves, we are going to be responsible for creating the ID3DXSkinInfo object. Of course, we will also have to populate it with skinning information before passing it along with the mesh into the CreateMeshContainer function.

So what does an ID3DXSkinInfo object contain? Well, when we create a new object of this type using the global D3DX function D3DXCreateSkinInfoFVF, we must pass in the total number of bones that influence the mesh. In our case, this will be the number of bones created for nodes belonging only to this branch. We must also inform the creation function about the number of vertices in the mesh which this object will contain skinning information for.

Once this function returns, we will have an ID3DXSkinInfo object that will have an empty table of bone slots. Each row of this internal table will eventually need to store the information for one bone that influences the mesh. For example, if we have a branch that uses five bones, the empty ID3DXSkinInfo would have a table with five slots. Each element of this table will be used to store a bone name, its bone offset matrix (which we will have to calculate) and an array of vertex indices describing which vertices in the branch mesh are influenced by this bone. It will also contain an array of weights describing the strength at which the bone influences the transformation of each vertex referenced in the bone's vertex index array.

Ultimately, this function will require allocating a new ID3DXSkinInfo for N bones, and then looping N times and setting the bone name, offset matrix, vertex indices and vertex weights for each bone using the following methods of the ID3DXSkinInfo interface:

```
HRESULT SetBoneName( DWORD Bone, LPCSTR pName );

HRESULT SetBoneOffsetMatrix( DWORD Bone,
                             const D3DXMATRIX *pBoneTransform );

HRESULT SetBoneInfluence( DWORD Bone,
                          DWORD numInfluences,
                          CONST DWORD *vertices,
                          CONST FLOAT *weights );
```

The three methods shown above are the only ones we will need to populate the ID3DXSkinInfo with the information about each of the branch bones. Remember, we are only interested in the bones that influence this mesh, which is a single branch. Since the BuildSkinInfo function will only be called for BRANCH_BEGIN nodes (and passed a pointer to that node), we simply have to traverse through the child nodes of the branch, adding bones to the ID3DXSkinInfo object as we encounter them between the start and end nodes of the branch.

The SetBoneName method is used to set the name of the bone we are currently processing. The first parameter describes the zero based index of the bone along the branch and the second parameter is where we pass in a string containing the name we would like to assign to the bone. Obviously this should match the name that we gave to the actual bone/frame in the hierarchy. For example, if this is the bone at the start node of the branch, this will have an index of zero and the name passed will match the name of the frame we created for this node and attached to the hierarchy. Now you know why we stored a frame pointer in the branch node structure when we created the bone hierarchy; it allows us to easily access the bone created from a given branch node here in this function.

As you undoubtedly remember from the previous chapter, the ID3DXSkinInfo object will also store a bone offset matrix for each bone in the hierarchy. We will need to calculate the bone offset matrix ourselves. As discussed in Chapter 11, the offset matrix for a given bone is used to transform its attached vertices into the space of that bone so that local rotation of the vertices about that bone can be achieved when combined with the absolute bone matrix. The bone offset matrix is usually calculated by taking the inverse of the absolute bone matrix in its default pose. This way, when the bone matrix is rotated to some degree and is combined with the bone offset matrix, we are essentially subtracting the reference pose bone matrix from the new absolute bone matrix, leaving us with only the relative rotation to apply.

Normally, when loading a skeletal construct from an X file, the frames of the hierarchy will compose a single skeleton (a character for example). As such, the bone offset matrix for each bone is simply calculated by traversing the hierarchy from the root frame and combining matrices as we step through the levels of the hierarchy, generating the absolute (not parent relative) bone matrix for each frame. We can then invert this matrix and have a matrix that will transform that bone and its vertices into a space where the bone itself is at the origin of the coordinate system during the transformation of the vertex by that bone (i.e., bone local space.)

Things are slightly different in our tree case because our frame hierarchy does not represent the skeleton of a single mesh. Rather, our hierarchy represents multiple mini-skeletons connected together, where each mini-skeleton is the skeleton for a single branch mesh. When calculating bone offset matrices, we are only interested in transforming the bone (and any influential vertices) back to the position of the root

bone of the skeleton for the current mesh. Therefore, we do not wish the bone offset matrix we generate for a bone to be the inverse of the complete concatenation of parent-relative matrices from the root frame of the entire hierarchy right down to that bone. Instead, we wish the bone offset matrix to be only the inverse of the concatenation of matrices from the bone that begins that branch (the root bone of the branch) to the current bone being processed. In short, when calculating the bone offset matrices, we will only be interested in the skeleton of the current branch mesh and not all the bones in the entire tree. Thus, this function will traverse through the child nodes of the branch (starting at the branch start node) and at each bone it visits, concatenate the relative matrices before passing the result down to the child. For any given bone, this matrix will contain the absolute transformation of the bone in the reference pose. All we have to do is invert this matrix and we have the bone offset matrix for the current bone being processed. We can then assign it to the bone using the ID3DXSkinInfo::SetBoneOffsetMatrix method.

The next step is collecting the indices of all vertices that are influenced by the bone we are currently processing so that we can send them into the ID3DXSkinInfo::SetBoneInfluence function. This function accepts as its first parameter the index of the bone we wish to set the vertex influences for. This will be the bone we are currently processing. We also inform this function about the number of vertices it will influence, which we will count when collecting those vertices (more on this in a moment). As the final two parameters to this function we send in pointers to two arrays. The first is the array of vertex indices describing the vertices influenced by this bone. The second is an array of weight values describing how strongly the bone influences the vertices in the previous array. As you will see in a moment, all vertices are influenced by only one bone in our branch mesh, so we will always pass in an array filled with a weight value of 1.0 for each vertex.

Additionally, since no vertex will ever be influenced by more than one bone and the vertices will be added to the ID3DXSkinInfo in the same order that they were added to our CTriMesh when traversing the tree, the indices we pass into this array will always have a 1:1 mapping with the order in which the vertex data was added to the mesh. For example, if the first bone influences the first 3 rings of vertices (starting from the branch start node) and the branch resolution is 8, when setting the first bone we would pass in an array of 8*3=24 indices where the indices in this array range from 0 to 23. Similarly, if the second bone in the branch influences the following 3 rings of vertices, we will be passing another 24 indices into this array when adding the second bone's data, which goes from 24 to 47, and so on. The number of rings of vertices that are influenced by a bone is the same for all bones (except the last). This value is always Bone_Resolution*Branch_Resolution. In other words, if the bone resolution is set to 3 then we know this means that, starting from the beginning of the branch, a bone will be inserted every 3 nodes. A bone is always inserted at the branch start node. Therefore, all rings of vertices inserted from that bone node up to, but *not* including the next bone node, are considered to be influenced by that bone. If the bone resolution was 3 and the branch resolution was 8 this means that every bone would influence 3 rings of vertices (including the vertices inserted at the bone node). If the branch resolution was 8, then each bone would influence 8*3 = 24 vertices. Therefore, we would have to assign 24 vertex indices and 24 weights (all set to 1.0) to each bone we create. The exception is the final bone in the branch which may have ended prematurely by some random calculation during the tree calculation process and because the final node will contain just a single tip vertex.

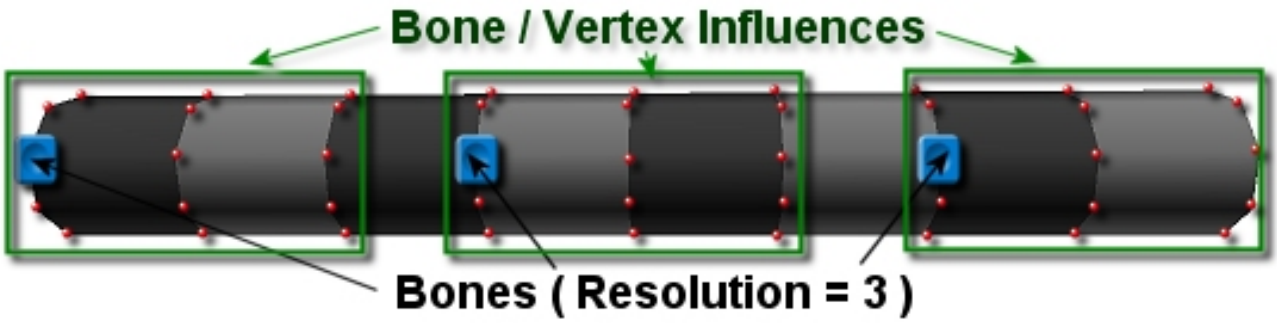

Figure 12.49 : Bone Vertex Influences for : Bone Resolution=3

Figure 12.59 shows the section of a branch where the bone resolution is set to 3. The blue boxes are the bones that are positioned every three ring's of vertices. The green boxes show the vertices which will be mapped to the bone and the bone's influence range over the branch.

Looking at this diagram and remembering that the pNode parameter passed into the BuildSkinInfo function will always be a node of type BRANCH_BEGIN, our strategy becomes clear:

1. Calculate the total number of vertices and bones in the branch.
2. Use the information from step 1 to allocate a new ID3DXSkinInfo of the correct size.
3. Step through each node in the branch starting at the branch start node.
    a. If this node is a bone node then add the name of the bone to the ID3DXSkinInfo and calculate its bone offset matrix. This will be an identity matrix for the start node of the branch.
    b. Add the indices of vertices at this node to a temporary index array.
    c. Step down into child node.
        i. If this node is a bone node, then assign all currently collected vertex indices which reference the previous bone created in step 3a, to the ID3DXSkinInfo object. Empty the temporary vertex index array.
        ii. If this is not a bone node then append the ring of vertex indices at this node to the temporary vertex index array.
4. Repeat steps 3a through 3c until last node is processed

As outlined above, we need to step through the nodes of the branch starting at the root **branch** node. When we find a bone node, we add this bone's information (name and offset matrix) to the ID3DXSkinInfo object. We then continue to traverse until the next bone is reached, collecting vertex indices in a temporary buffer along the way. As soon as we hit a new bone node, we know that the vertex indices currently stored in our temporary array belong to the previous bone, so we call the SetBoneInfluence method to assign them to that bone. We then empty the temporary vertex array so that it can be used to collect indices between the next two bones. We create a new bone at the current node and continue the process.

Let us now look at the code one section at a time. The function accepts three parameters that are passed to it by the BuildNode function. The first is a pointer to the BRANCH_BEGIN node of the branch mesh we are about to calculate the bone influences for. The second is a pointer to the branch mesh, which at this point contains all its vertex and index data. The third parameter is the address of an ID3DXSkinInfo interface pointer which on function return should point to a valid ID3DXSkinInfo interface containing

the bone influences for the mesh. The BuildNode method can then pass this interface into the CreateMeshContainer function for final skinning.

```
HRESULT CTreeActor::BuildSkinInfo( BranchNode *pNode,
                                   CTriMesh * pMeshData,
                                    LPD3DXSKININFO *ppSkinInfo )
{
    HRESULT hRet;
    ULONG  BoneCount   = 0, InfluenceCount=0, Counter=0,IndexCounter = 0, i;
    ULONG  VertexCount = pMeshData->GetNumVertices();

    ULONG      * pIndices    = NULL;
    float      * pWeights    = NULL;
    BranchNode * pSearchNode = NULL;
    BranchNode * pSegmentNode= NULL;
    TCHAR        strName[1024];
    D3DXMATRIX   mtxOffset, mtxInverse;

    // Set offset to identity.
    D3DXMatrixIdentity( &mtxOffset )
```

In the above code we calculate the number of vertices in the mesh and store it in the VertexCount parameter. We will need to know how many vertices our branch mesh contains so that we can feed it into the ID3DXCreateSkinInfoFVF function. We also initialize the local matrix variable (mtxOffset) to an identity matrix. This will be used to store the bone offset matrix for each bone we add to the ID3DXSkinInfo object. However, as the first bone we will add will be the bone at the branch start node, this is the base frame of reference for the entire mesh and the bone offset matrix should be an identity matrix.

In the next section of code we will use the pSearchNode pointer to count all the bones we have added to this branch in the hierarchy. We will need to supply D3DX with this information when it creates the ID3DXSkinInfo object.

Note that the current node may have multiple children if other branch nodes begin at the next node. That is, the next node in the current branch might be in a sibling list with multiple BRANCH_START nodes which we are *not* interesting in processing. Therefore, to locate the next child node we will step down to the first child node of the current node and then search for the one (and only) node in that list that is not a BRANCH_BEGIN node. That is the node in that sibling list that is the continuation of this branch and the next node we must process. Essentially, the next section of code steps through every node in the branch looking for bone nodes (nodes that had bones attached to them) and for each one it finds it increments the BoneCount local variable. This will allow us to count all the bones used by this mesh so that we can allocate the ID3DXSkinInfo to manage that many bones.

```
    // Navigate our way through the branch, setting bone details
    BoneCount   = 1;
    pSearchNode = pNode;

    while ( pSearchNode = pSearchNode->Child )
    {
        // We're not interested in other branches, so shift us through until
```

```
        // we find a node that is our segment, or end node.
        while ( pSearchNode->Type == BRANCH_BEGIN && pSearchNode )
                pSearchNode = pSearchNode->Sibling;

        // If we couldn't find a node, we have an invalid hierarchy
        if ( !pSearchNode ) return D3DERR_INVALIDCALL;

        // Was a frame created here?
        if ( pSearchNode->BoneNode ) BoneCount++;

    } // Next child segment
```

At the start of the above code we initially set the bone count to 1. This is because the first node is a branch start node and will always have a bone. Also, when we enter the while loop that steps through the child branch nodes, we start at the child of the BRANCH_BEGIN node. Therefore, the initial bone at the start of the branch would not be accounted for when keeping the count. Notice that with each iteration of the outer while loop, we step down one level in the hierarchy into the current node's sibling list. The inner while loop searches that sibling list for the only non-BRANCH_BEGIN node, if multiple nodes exist in that list. Essentially, the above code is just stepping down the branch each time, finding the next child node that continues the current branch, and notching the score up for each bone node that is found.

We now have everything we need to allocate a new ID3DXSkinInfo object. We know the number of bones that influence our mesh and the number of vertices in this mesh.

```
    // Create the skin info object ready for building.
    hRet = D3DXCreateSkinInfoFVF(VertexCount, VERTEX_FVF, BoneCount, ppSkinInfo );
    if (FAILED(hRet)) return hRet;

    // Get dereferenced skin info for easy access
    LPD3DXSKININFO pSkinInfo = *ppSkinInfo;
```

When the D3DXCreateSkinInfoFVF function returns, the ID3DXSkinInfo interface pointer (the final parameter) will point to a valid interface. In this case, we pass in the ID3DXSkinInfo interface pointer that was passed into the function by BuildNode so that it will be able to access it on function return. We then assign this interface pointer to a local variable for ease of access so we do not have to de-reference the original pointer each time.

At this point we have our new ID3DXSkinInfo interface but it contains no data. It has empty slots where the bone information should be, so we will spend the remainder of this function setting the information for each bone in the branch mesh.

As mentioned previously, we will need to step through the nodes of this branch and collect the indices of all vertices that are influenced by the bone we are processing. All vertices between two bones will be influenced by the previous bone (see Figure 12.49). Therefore, while stepping between two bones, we will need a temporary vertex indices buffer where we can store the data we have collected at each node as we traversed from one bone to the next. We will also need a temporary array of the same size to store the weights of each of these vertices (one weight for each vertex). We allocate these temporary buffers to be large enough to hold all the vertices in the mesh, just to be safe (e.g., if one bone in the mesh

directly influenced all its vertices, we will still be covered). The next code block allocates the vertex index array and the vertex weights array.

```
    // Allocate enough space to hold all indices and weights
    pIndices = new ULONG[ VertexCount ];
    if ( !pIndices) return E_OUTOFMEMORY;

    pWeights = new float[ VertexCount ];
    if ( !pWeights ) { delete []pIndices; return E_OUTOFMEMORY; }
```

We will step through the nodes of this branch using a while loop, but this loop will start at the child node of the branch start node. Therefore, before we enter the loop we will give the ID3DXSkinInfo object the information (vertices and weights) for the bone stored at the branch start node. We know that the branch start node will always have a bone and that the bone will always by the first bone in the mesh (index 0).

In the next section we setup the first bone in the ID3DXSkinInfo object. We build the name of this bone in the same way we built the name of the bone when we added it to the hierarchy. We must make sure that the name we add here matches the name of the bone in the frame hierarchy it represents. Once we have the name stored in a string, we set it along with the bone offset matrix for this bone (identity for the branch start bone) as shown below.

```
    // Generate root bone name (matches frame)
    _stprintf( strName, _T("Branch_%i"), pNode->UID );

    // Set it to the skin info
    pSkinInfo->SetBoneName( Counter, strName );
    pSkinInfo->SetBoneOffsetMatrix( Counter, &mtxOffset );
    Counter++;
```

The first parameter to both the SetBoneName and SetBoneOffsetMatrix functions is the index of the bone we wish to set. Since the branch start bone is the first bone in the branch it should have an index of 0. This is the initial value of the Counter variable which was initialized at the top of the function. We then increment Counter so that when we add another bone later we know this next bone will be at index 1 in the ID3DXSkinInfo object's internal bone table.

Now we need to add the indices and weights for the vertices that are influences by this bone. We will collect this information in a while loop starting from the child node of the branch start node. This means we must make sure that we add the first ring of vertices that were inserted at the branch start node so that they are not forgotten. The next section of code loops around every vertex in the first ring (Branch_Resolution) adding the index of each vertex to the pIndices temporary array and a value of 1.0 in the corresponding temporary pWeights array.

```
    // Fill in the values so far for the root of the branch
    for ( i = 0; i < m_Properties.Branch_Resolution; ++i )
    {
        pIndices[ InfluenceCount ] = IndexCounter++;
        pWeights[ InfluenceCount ] = 1.0f;
        InfluenceCount ++;
    } // Next index
```

The IndexCounter variable will start off at zero and be incremented for *every* vertex we add; it is not reset when we encounter a new bone. If our mesh has 400 vertices, then we know that the order in which we added them to the mesh is the same order in which they are assigned to their bones. Therefore, we will essentially add vertices 0-399 in exactly that order. Of course, here, we are just adding the first ring of vertices. If the branch resolution was 8 for example, we would add the values 0-7 to the indices array. When we visit the next child node and add that ring of vertices, the index counter will have not been reset, so we will be adding values 8 through 15 to the indices array, and so on.

What is InfluenceCount in the above code? In the pIndices array we are collecting the influenced vertex indices for one bone. Therefore, influence count will be reset to zero each time we start collecting vertices for a new bone. It is simply used to place the vertex index in the correct zero based location in the indices array for the current bone. For example, if a bone is influenced by 40 vertices, while collecting those vertices, influence count will count from 0 to 39. When a new bone is encountered, we will add all currently collected vertices to the ID3DXSkinInfo for the previous bone, reset the InfluenceCount back to zero, and start collecting vertex indices all over again using the same buffer for the next bone (overwriting any previous data).

At this point we have added the first bone's name and offset matrix and have collected the first ring of vertices from the BRANCH_BEGIN node and stored them in the indices buffer. We will now enter a while loop that will start at the child node and continue to make its way down to the end of the branch. It will collect the vertices from each node exactly as we did above and add them to the indices buffer. When we finally hit another bone node, we know that the indices buffer will contain all the vertex indices influenced by the previous bone, so we can add them to the ID3DXSkinInfo and attach them to the previous bone. We then flush the indices buffer, reset the influence count and start collecting vertices for this new bone, and so on.

Let us now examine the while loop which comprises the rest of the function and adds the remaining bone information for the branch.

As before, we start by stepping down into the child node of the BRANCH_BEGIN node. This is the first node we process in this loop. We then use the same inner while loop mechanism we used before to make sure that if the child node is in a sibling list with other nodes, we locate the correct node that is the continuation of the branch (not a BRANCH_START that spawns from that node).

```
    // Now we make our way through and build the skin info
    pSearchNode = pNode;
    while ( pSearchNode = pSearchNode->Child )
    {
      // Reset the segment node (we must find one at each level)
       pSegmentNode = NULL;

       // Loop through all siblings of this node
       for ( ; pSearchNode; pSearchNode = pSearchNode->Sibling )
       {
           // If this is a begin node, skip it. We aren't interested in other branches
           if ( pSearchNode->Type == BRANCH_BEGIN ) continue;

           // If this is the segment node, store it, we want to continue down from here
           if ( pSearchNode->Type == BRANCH_SEGMENT ) pSegmentNode = pSearchNode;
```

If we find any node in the sibling list that is a BRANCH_BEGIN node, we skip it and advance to its sibling. We are searching for the node that is a continuation of the current branch; the only node in the sibling list which is of type BRANCH_SEGMENT.

If we get this far then pSearchNode points at the child node (i.e., the next node we will process). We also store a copy of this node's pointer in the pSegmentNode local variable for later use.

The first thing we will do is test to see if this is a bone node. If it is, then it is time to calculate the bone offset matrix for this new bone, build its name, and set the name and bone offset matrix in the next row of the ID3DXSkinInfo bone table.

```
        // Was a frame created here?
        if ( pSearchNode->BoneNode )
        {
            // Combine the bone matrix, and inverse for the reference pose offset
            D3DXMatrixMultiply( &mtxOffset,
                                &pSearchNode->pBone->TransformationMatrix,
                                &mtxOffset );

            D3DXMatrixInverse( &mtxInverse, NULL, &mtxOffset );

            // Generate bone name (matches frame)
            _stprintf( strName, _T("Branch_%i"), pSearchNode->UID );

            // Set it to the skin info
            pSkinInfo->SetBoneName( Counter, strName );
            pSkinInfo->SetBoneOffsetMatrix( Counter, &mtxInverse );

        } // End if bone created
```

Notice that every time we encounter a bone we combine its parent relative matrix (the frame matrix) with the current contents of the mtxOffset matrix (which will be set to identity at the start node of the branch). This means that whenever we reach a bone, it will always contains the absolute transformation of that bone in its reference pose relative to the start node of the branch. As discussed earlier, inverting this matrix gives us our bone offset matrix. Also remember that when we added the first bone we incremented the Counter variable, so if this was the first bone we encountered after the branch start node, Counter would be set to 1 and we would be setting the properties of the second bone in the ID3DXSkinInfo bone table.

As discussed, when a new bone is encountered, we need to take the current vertex indices we have collected between the previous bone and this new bone and assign them as influences for the previous bone. We also have to perform this same step if the current node is a BRANCH_END node as we have reached the end of the line. In the case when the node is an end node, we also add the final tip vertex index to the index array before we assign the influences.

```
        // If we're about to switch to a new bone, or we have completed
        // this branch, fill in the PREVIOUSLY started bone's influence details
        if ( pSearchNode->BoneNode || pSearchNode->Type == BRANCH_END )
        {
            // If this is the end of the branch, just add the last influence
```

```
            if ( pSearchNode->Type == BRANCH_END )
            {
                // There is only one tip vertex on branch ends
                pIndices[ InfluenceCount ] = IndexCounter++;
                pWeights[ InfluenceCount ] = 1.0f;
                InfluenceCount++;

            } // End if end

          // Set the bone influence details
          pSkinInfo->SetBoneInfluence(Counter-1,InfluenceCount,pIndices,pWeights );

            // Reset to start again
            InfluenceCount = 0;

            // Move on to next bone
            Counter++;

        } // End if switching
```

Notice that when we call SetBoneInfluence function we use the index Counter - 1 to assign everything accumulated to the previous bone. Because Counter always contains the index of the next bone we hope to encounter, subtracting 1 from it gives us the index of the previous bone we processed. This is the bone which the vertices we have accumulated in the pIndices buffer should be assigned to. When we add the new bone and assign the collected indices to the previous bone, we reset InfluenceCount to zero to start collecting indices at the beginning of the array again for the next bone. We also increment the Counter variable so that we know the index of the next bone we encounter and which slot it should occupy in the ID3DXSkinInfo object's bone table.

The final section of code shows what happens when the node is not a bone node or branch end node. If for example, your branch had a bone resolution of 5, the code in the above code block would only be executed every 5 nodes of the branch. For all nodes in between bone nodes, the following code would be executed. It simply adds the vertex indices for the next ring of vertices to the pIndices buffer.

```
        // Add influences to the previous bone
        if ( pSearchNode->Type == BRANCH_SEGMENT )
        {
            for ( i = 0; i < m_Properties.Branch_Resolution; ++i )
            {
                pIndices[ InfluenceCount ] = IndexCounter++;
                pWeights[ InfluenceCount ] = 1.0f;
                InfluenceCount ++;

            } // Next index

        } // End if segment

       } // Next Node Sibling

       // If we couldn't find a segment node in the sibling list
       // we've reached the end of the branch
       if ( !pSegmentNode ) break;
```

```
    } // Next child segment

    // Clean up
    delete []pIndices;
    delete []pWeights;

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

At the end of the function you can see that we release the temporary index and weight arrays and return.

We have now covered the entire tree generation process and program flow returns to CTreeActor::GenerateTree. If we look at the code for that function again we can see that there is only one more task that must be performed before program flow goes back to the caller (the application in our case) and the tree is considered complete.

```
HRESULT CTreeActor::GenerateTree( ULONG Options, LPDIRECT3DDEVICE9 pD3DDevice,
                                  const D3DXVECTOR3 &vecDimensions,
                                  const D3DXVECTOR3 &vecInitialDir,
                                  ULONG BranchSeed )
{
    HRESULT            hRet;
    CAllocateHierarchy Allocator( this );

    // Validate parameters
    if ( !pD3DDevice ) return D3DERR_INVALIDCALL;

    // Release previous data.
    Release();

    // Store the D3D Device here
    m_pD3DDevice = pD3DDevice;
    m_pD3DDevice->AddRef();

    // Store options
    m_nOptions = Options;

    // Generate the branches
    hRet = GenerateBranches( vecDimensions, vecInitialDir, BranchSeed );
    if (FAILED(hRet)) return hRet;

    // Build the frame hierarchy
    hRet = BuildFrameHierarchy( &Allocator );
    if (FAILED(hRet)) return hRet;

    // Build the bone matrix tables for all skinned meshes stored here
    if ( m_pFrameRoot )
    {
        hRet = BuildBoneMatrixPointers( m_pFrameRoot );
        if ( FAILED(hRet) ) return hRet;

    } // End if no hierarchy
```

```
    // All is well.
    return D3D_OK;
}
```

Just as we did in CActor::LoadActorFromX, as soon as the mesh hierarchy has been constructed we must call the base class BuildBoneMatrixPointers function. Hopefully you will recall from Lab Project 11.1 that this function traverses the entire hierarchy searching for mesh containers. For each mesh container found that stores an ID3DXSkinInfo object, it extracts each of the bone names, searches for the matching frame in the hierarchy, and adds the pointer to each frame's absolute matrix to the mesh container's bone matrix pointer array. When this function returns, each mesh container will contain an array of bone matrix pointers and an array of matching bone offset matrices that can be easily accessed and combined when rendering the mesh.

12.5 Animating CTreeActor

The CTreeActor class includes a function that creates animation data for the frame hierarchy. The GenerateAnimation function should be called by the application after the call to the GenerateTree function. This function is not automatically called by GenerateTree since you might want to generate a tree that does not animate, or perhaps you want to animate the tree using your own algorithms. Furthermore, if you intend to simply save the tree data out to disk and import it into GILES™ for placement purposes, the animation data will be lost anyway.

However, we felt that it would be helpful to go through the process of creating some simple animations. It should provide you with some additional insight into the animation system as a whole and hopefully spark some of your own creative ideas regarding animation. If you wish to use the feature, the GenerateAnimation function can be called to build keyframes for each bone in the tree. The animation we will create will simulate wind and the tree branches will sway back and forth.

To get an idea of the overall process, below we see how some code may look in an application that has created a CTreeActor and would like it to animate.

```
// Allocate new Tree Actor
CTreeActor *pTree = new CTreeActor;

// Set actor skinning mode
pTree->SetSkinningMethod( CActor::SKINMETHOD_AUTODETECT );

// Register global texture/material callback for non-managed mode
pTree->RegisterCallback( CActor::CALLBACK_ATTRIBUTEID, CollectAttributeID, this );

// Set the texture and material to use for the branches
pTree->SetBranchMaterial( _T("LondonPlaneBark.dds"), NULL );

// Generate skinned tree
pTree->GenerateTree( D3DXMESH_MANAGED,
                     m_pD3DDevice,
                     D3DXVECTOR3( 2.0f, 2.0f, 2.5f ),
```

```
                    D3DXVECTOR3( 0.0f, 1.0f, 0.0f ));

// We wish to animate so generate animation data and controller for actor
pTree->GenerateAnimation( D3DXVECTOR3( 0.0f, 0.0f, 1.0f ), 30.0f, true );
```

For completeness, we see the actor being allocated and its skinning method being set to auto-detection mode. We then see the scene registering a callback function that will be used by the actor to process the textures and materials needed by the actor. The application then calls the SetBranchMaterial method to supply the actor with the filename of the texture it would like mapped to its branches. Notice that the second material parameter, which can be used to pass a pointer to a D3DMATERIAL9 structure, is set to NULL in this example. We have elected to let the tree use the default material it creates in its constructor.

The call to the GenerateTree method is then made with the final two vectors describing the dimensions of the root node and the direction vector for that node. When this function returns, the actor will have a final frame hierarchy and every branch will be represented by a skinned mesh.

On the final line we call the CTreeActor::GenerateAnimation method, which will create the actor's animation controller and populate it with animation data. The function takes three parameters. The first is the wind direction vector which will be used to calculate how the bones of the tree will animate. This vector describes the direction that the wind is blowing and is used to calculate a rotation axis for the bones. The second vector is the strength of the wind, which is used to scale the rotation of each branch with respect to the rotation axis. The third parameter is a boolean which specifies whether the CActor::ApplyCustomSets function of the base class will be called after the sets have been created. As you will recall, this function converts each ID3DXKeyframedAnimationSet into one of our custom CAnimationSets to overcome the GetSRT bug. Since the GenerateAnimation function will only create one animation set for the entire tree, only this one animation set will have to be converted in the ApplyCustomSets method. However, we have left this parameter as a Boolean so that should this bug be fixed in the future, you can pass false and use the ID3DXKeyframedAnimationSets that are initially created. Lets us now examine the code to this function and discuss the animation it is trying to implement.

GenerateAnimation - CTreeActor

The animation we will create will be very simple. Given a wind direction vector, we will perform the cross product between that vector and the world up vector <0, 1, 0> to get a vector that is perpendicular to the wind direction vector. We can think of this new vector as being a rotation axis which the direction vector is blowing directly against. If we create a scenario where the axis exists at each bone in the tree, we can rotate the branches about this axis by a small amount determined by the fWindStrength parameter passed into the function.

By default, we will create 20 keyframes for each bone in the hierarchy. For each of those keyframes (for a given frame) we will rotate the frame by some amount. Each keyframe for every frame will contain a rotation around the same axis, but at varying degrees. The pattern we want to simulate is not simply for the branches to always be blowing away from the wind, as this would simply bend the tree away from

the wind direction vector. Rather, we wish to rotate the bones away from the wind vector but also let them move back again so that the branches sway back and forth. Admittedly, this is not particularly realistic, nor is it going to be physically correct, but the end result will be close enough to what we want to work for our purposes in this lesson.

As an example of how things will work, the animation stored in each keyframe for a given frame might slowly rotate the bone an angle of 30 degrees away from the wind vector around the rotation axis in keyframes 1 – 10. It might then spend the next 10 keyframes rotating it back again to its initial position. Our animation will not be quite that simplistic, but hopefully you understand that each frame will have an array of 20 keyframes and that at each keyframe, we store the rotation angle of the bone around the rotation axis at that particular time. To make things more realistic, we will also introduce the concept of a height delay, so that the further up the tree we move, the bones rotate slightly behind the bones beneath them in the tree. This will generate a whip-like effect, where the bottom of branches will start to move first and the end of the branch will be dragged along with it a short time later. At the midway point, the bottom section of a branch may be rotating back to its initial pose while it top part is still on its way out to full extension (the position at which the bottom section starts to pull the top section of the branch back with it).

The actual creation of the keyframe data is done using a recursive function called BuildNodeAnimation. It is called from the GenerateAnimation function to traverse all the nodes in the tree and build the keyframes for each one. We will discuss this all shortly. First though, let us have a look at the parent function and the doorway to this recursive process.

One of the first things this function must do is create the actor's ID3DXAnimationController. Because we did not load our actor's data using the D3DXLoadMeshHierarchyFromX function, our actor will currently have no animation controller assigned. However, before allocating this new controller, the function will release the previous controller interface if one exists. This is done for safety as it lets the application call this function even if the actor has previously had animation defined for it. It is possible that the application may wish to call this function again to generate animation data with a different wind direction vector or wind strength than that specified the first time the function was called.

Once we have released the controller interface we must create a new one. Recall that when we use the D3DXCreateAnimationController function to create a new controller we must pass in the limits. We only need one animation set and we may as well specify the default number of tracks (2) and the default number of key-events (30) even though we do not need them.

However, you will recall from our discussion of the animation controller in Chapter 10, that when creating one, we must know how many animation outputs (i.e., frame matrices) to make room for inside the controller. Luckily, we can use the D3DXFrameNumNamedMatrices helper function to traverse the actor's frame hierarchy and count the number of named frames. We will then reserve this many animation outputs so that we reserve enough room for every frame in the hierarchy to be manipulated by the animation controller. Furthermore, once we have created the animation controller, we still have to register each frame matrix in our hierarchy with the controller so that the controller can cache a pointer to it for animation updates. We can use the other D3DX global function D3DXFrameRegisterNamedMatrices function to automate that process too. We simply pass this function the root frame of our hierarchy and it will traverse it and register the parent-relative matrix for

each named frame as an animation output in the controller. The first section of the function that performs these tasks is shown next.

```
HRESULT CTreeActor::GenerateAnimation( D3DXVECTOR3 vecWindDir,
                                       float fWindStrength,
                                       bool bApplyCustomSets /* = true */ )
{
    HRESULT                     hRet;
    D3DXTRACK_DESC              Desc;
    ULONG                       FrameCount;
    D3DXVECTOR3                 vecWindAxis;
    LPD3DXKEYFRAMEDANIMATIONSET pAnimSet = NULL;

    // If there is already an animation controller, release it. This allows
    // the application to call this function again, specifying different
    // properties without having to rebuild the tree itself.
    if ( m_pAnimController ) m_pAnimController->Release();
    m_pAnimController = NULL;

    // Count the number of frames in our hierarchy
    FrameCount = D3DXFrameNumNamedMatrices( m_pFrameRoot );

    // Create a new animation controller
    hRet = D3DXCreateAnimationController(FrameCount,1,2,30,&m_pAnimController );
    if ( FAILED(hRet) ) return hRet;

    // Register all of our frame matrices with the animation controller
    hRet = D3DXFrameRegisterNamedMatrices( m_pFrameRoot, m_pAnimController );
    if ( FAILED(hRet) ) return hRet;
```

We now have an empty animation controller that has room for one animation set. Let us now create that animation set and call it ‘SwaySet’ (it is the only set we will use). We will work using a ticks-per-second ratio (timing resolution) of 60 for our keyframe timestamps and set the animation to loop.

```
    // Create a new animation set
    hRet = D3DXCreateKeyframedAnimationSet( _T("SwaySet"),
                                            60.0f,
                                            D3DXPLAY_LOOP,
                                            FrameCount,
                                            0,
                                            NULL,
                                            &pAnimSet );
    if ( FAILED(hRet) ) return hRet;
```

The fourth parameter in the above code is where we specify the number of animations we would like to store in this animation set. Remembering that an Animation is a collection of keyframes for a single frame in the hierarchy, we know that we wish to have an animation for every hierarchy frame. Therefore we pass in the value FrameCount, which is the total number of frames in our hierarchy as calculated earlier in the function. The fifth and sixth parameters are set to zero and null as we have no need to register callback keys. As the final parameter, we pass the address of an ID3DXKeyframedAnimationSet interface pointer which, on function return, will point to our new and empty animation set. The animation set is still not registered with the controller at this point.

We now have an animation set that we will use to store the keyframes for each frame in the hierarchy. Before we calculate those keyframes however, we need to know the axis of rotation we will use for the bones of our tree. The wind direction vector was passed into the function and describes the exact direction the wind is blowing in tree space. As such, we know that this vector is perpendicular to the axis we actually wish to rotate about. For example, imagine a player mounted on a pole in a table football game (also called Foosball in some places). The wind direction can be thought of as the vector hitting the player head on, while the pole on which the player is mounted is actually the rotation axis the player will rotate around in response to that wind. Therefore, provided our wind direction vector is not equal to the world up vector <0,1,0>, which should never realistically be the case, we can cross these two vectors to get the rotation axis by which we will rotate each bone.

```
    // To make things a little more user friendly we accept a wind direction to
    // this function, however BuildNodeAnimation requires an axis about which
    // the branches will rotate. We convert this here.
    D3DXVECTOR3 vecCross = D3DXVECTOR3( 0.0f, 1.0f, 0.0f );
    D3DXVec3Cross( &vecWindAxis, &vecCross, &vecWindDir );
```

We now have the rotation axis stored in the vecWindAxis member, so it is time to walk through the nodes of the tree and generate the keyframe data. This step is handled with a single call to the recursive function BuildNodeAnimation, whose code we will discuss next. This function is passed the root node of the branch node hierarchy, the rotation axis we just calculated, the wind strength, and the interface of our new animation set, which will be the recipient of the keyframe information calculated by this function.

```
    // Build the animations
    hRet = BuildNodeAnimation( m_pHeadNode, vecWindAxis, fWindStrength, pAnimSet);
    if ( FAILED(hRet) ) { pAnimSet->Release(); return hRet; }
```

When this function returns, every frame in the actor's hierarchy will have had its animation generated and added to the animation set. Next we must register this animation set with our animation controller and assign it to the first track on the mixer.

```
    // Register the animation set with the controller
    hRet = m_pAnimController->RegisterAnimationSet( pAnimSet );
    if ( FAILED(hRet) ) { pAnimSet->Release(); return hRet; }

    // Set this in track 0 to ensure it plays
    SetTrackAnimationSet( 0, pAnimSet );
```

We must be careful to initialize the properties of any track we use, especially when creating the animation controller manually. D3DX will not perform any track property initialization (we learned the hard way in our tests that the initial properties of a track are all zero -- which generates no visual output). In the following section of code we set the properties of the first and only mixer track we will use such that it is enabled, has a speed of 1.0 (the default), a weight of 1.0 (the default) and has a track position of 0.0 (the start).

```
    // Setup the track description (defaults to disabled)
    ZeroMemory( &Desc, sizeof(D3DXTRACK_DESC) );
```

```
    Desc.Enable   = true;
    Desc.Weight   = 1.0f;
    Desc.Speed    = 1.0f;
    Desc.Position = 0.0f;

    // Set the track description
    m_pAnimController->SetTrackDesc( 0, &Desc );
```

At this point we can release the pAnimSet set interface since it has been registered with the controller and had its reference count increased.

```
    // We can release the animation set now (controller owns it)
    pAnimSet->Release();
```

As a final step we see whether the application would like the ID3DXKeyframedAnimationSet we have just created replaced with our custom CAnimationSet object. If true is passed as this parameter, we call the base class method ApplyCustomSets, which we know from the previous lesson performs this task.

```
    // Apply custom animation sets if requested
    if ( bApplyCustomSets ) ApplyCustomSets();

    // Success!!
    return D3D_OK;

}
```

The function then returns to the application with a completely valid animation controller that can be used by the application to update the animation (via AdvanceTime).

There was nothing too challenging happening in that function, but then again, the real work is performed in the BuildNodeAnimation call. This function recursively visits every frame in the hierarchy, generating animation data for it. Let us have a look at that final function now.

BuildNodeAnimation - CTreeActor

When this function is first called, it is passed the root node of the branch node hierarchy. The function will recursively call itself visiting all child and sibling nodes in the tree. However, only significant action is taken when the node we are visiting is a bone node. If it is a bone node then we know this node has a corresponding frame in the hierarchy and it must have an animation built for it and registered with the passed animation set.

Virtually all the code in this function is placed within the “if(pNode->BoneNode)” code block. The only code that is not in that block comes at the end of the function when the function calls itself recursively if child and/or sibling nodes exist. Let us take a look at the first section of this function.

```
HRESULT CTreeActor::BuildNodeAnimation( BranchNode * pNode,
                                        const D3DXVECTOR3 & vecWindAxis,
```

```
                                        float fWindStrength,
                                        LPD3DXKEYFRAMEDANIMATIONSET pAnimSet )
{
    ULONG       i;
    HRESULT     hRet;
    D3DXVECTOR3 vecChildWindAxis = vecWindAxis;

    // If this is a bone node, build animation for us.
    if ( pNode->BoneNode )
    {
        D3DXKEY_VECTOR3    pTranslation[2];
        D3DXKEY_QUATERNION pRotation[20];
        D3DXQUATERNION     quatRotate, quatValue;

        // Transform the wind axis by this frame
        D3DXMATRIX  mtxInverse;
        D3DXMatrixInverse(  &mtxInverse,
                            NULL,
                            &pNode->pBone->TransformationMatrix );


        D3DXVec3TransformNormal(&vecChildWindAxis,&vecChildWindAxis,&mtxInverse );
```

The above code block is only executed for bone nodes. First it allocates space for two translation keyframes and 20 rotation keyframes. We have decided that each node will need 20 rotational keyframes to get good looking rotation results on screen. Although we never wish to actually translate any bone directly, unfortunately, we must always include two translation keyframes; one at the start one at the end. We do this is because the SRT data stored in the keyframes overwrites the data in the frame's transformation matrix when we call AdvanceTime. If no translation vectors are specified at all, then a translation vector of <0,0,0> will be assumed and will overwrite the actual parent-relative positional offset of the frame stored in its matrix.

For example, if we have a frame with a parent-relative matrix translation vector of <10,10,10>, we know that this offsets it from the position of the parent frame <10,10,10> units in parent space. However, when the animation has no translation or scale keyframes, <0,0,0> is used. In the case of our matrix, the <10,10,10> translation vector in the frame matrix would be overwritten each time with a translation value of <0,0,0>. This would essentially translate the child frame back to the position of the parent bone and cause a huge mess. As long as we supply at least two translation keyframes, the animation system will happily interpolate between them. The result of the interpolation will be used to replace the translation vector in the frame matrix.

Therefore, we extract the position vector stored in the frame matrix (<10,10,10> for example) and store that same position vector in a translation keyframe at the start of the animation (time 0) and a keyframe at the end of the animation (time = max time). Every time AdvanceTime is called by the application, the GetSRT function will interpolate between <10,10,10> and <10,10,10> and of course, generate the parent-relative position of <10,10,10>. Simply put, by inserting two translation keyframes at the start and end of the periodic position of the animation set, we retain the position vector of the frame in its default pose throughout the interpolation process. You will see us setting up these two translation frames in a moment.

Notice in the above code that we multiply the rotation axis (ChildWindAxis) by the inverse parent relative matrix of the current frame. This is because the rotation axis was originally defined in tree space and we need to perform the rotation in bone space. This is the same as multiplying the vector with the frame's bone offset matrix, although done slightly differently. In this case, we are multiplying it by the inverse of the parent relative matrix instead of using the inverse of the absolute matrix of the frame. However, notice the transformed rotation vector is passed by reference to its child node; therefore, instead of accumulating relative matrices and using the inverse as the bone offset matrix, we are instead accumulating the transformations applied to the wind vector from inverse parent relative matrices, which equates to the same thing. Suffice to say, at this point, we have the tree space wind vector transformed into the space of the bone's local coordinate system.

We do not want all nodes in our tree to rotate back and forth at the same time since this would look a little unnatural (like rotating a cardboard cut-out of a tree on a pivot). Instead, we would like the base of the tree to start rotating first. Then, the higher nodes in the tree will have a slight delay on them as if trying to catch up, but never quite doing so. Rather than a straight pivot motion, it causes a sweeping motion and the rotation applied to the lower branches is filtered slowly up to the branch end segments.

In order to accomplish this, we calculate a delay value that is a function of the current node's depth in the tree. By dividing the node's iteration value by the maximum iteration count of the virtual tree, we are essentially mapping the iterations of all nodes into the 0.0 to 1.0 range (much like when calculate the V texture coordinates of each vertex). So a node at the deepest level of the hierarchy (branch end nodes) is assigned a delay value of 1.0, and a node at the lowest level of the hierarchy is assigned a delay value of 0.0. Any nodes in between these hierarchy levels are mapped a value in the 0.0 to 1.0 range. Notice however, that we negate the calculation such that the branch end nodes will have a delay of –1 and the root would have a delay value of zero. Do not worry about how this is used at the moment, just be aware that the delay value describes the position of the node in the tree as a number from 0 (base of tree) to –1 (tips of branches).

```
        // Higher Up bigger animation delay causes a sweep motion
        float fDelay = -((float)pNode->Iteration /
                         (float)m_Properties.Max_Iteration_Count);
```

Not only do we need a delay such that bones positioned higher up the tree rotate slightly after bones placed towards the bottom of the tree, we also need another scalar that will be used to scale the amount of rotation we apply to a bone. If we think about a tree swaying in the wind, we know that thinner lighter branches experience much more motion than thicker heavier branches. So we can see that even if the base of the tree was swaying slightly from side to side by let us say –4 to +4 degrees, this same movement filtered up to the branch ends would result in much more movement. We might imagine the upper branches rotating between –15 and +15 degrees.

Thus, we will create a scalar in the 0.0 to 1.0 range that acts as a way to dampen the amount of rotation applied to each bone. Dampening will be a function of that node's depth in the branch node hierarchy. Once we calculate a rotation angle, it will be scaled by this value. A bone at the top of the tree would have no dampening (1.0) and the full angle value will be used in its rotation (Angle * 1.0). Bones at the base of the tree would have full dampening (Angle * 0.0). Bones in between these two extremes would be scaled appropriately. The nearer to the end of a branch a bone is located, the less dampening will

occur. We calculate this dampening value for the current node by dividing the position of the node in the hierarchy by the total depth of the hierarchy (much like above, only this time the value is not negated).

```
// The amount of motion applied at this node. Simple calculation which
//  increases the overall movement of the branch the higher up we get.
float fMovement = ((float)pNode->Iteration /
                   (float)m_Properties.Max_Iteration_Count);
```

This is a simple way to calculate the dampening factor, but not necessarily the most realistic. You might decide to include other factors in your implementation (actual branch thickness for example).

It is now time to loop around and build the 20 keyframes for this node. We will set the maximum time of the animation to run for 600 ticks. In this loop, each keyframe we add will be for a time between 0 and 600 ticks (evenly spaced) and will be a rotation keyframe.

```
// Retrieve the frame for the bone
D3DXFRAME_MATRIX * pFrame = (D3DXFRAME_MATRIX*)pNode->pBone;

// Generate rotation over n keys
ULONG KeyCount = 20;
float MaxTime  = 600.0f;

for ( i = 0; i < KeyCount; ++i )
{
    // Set the key time
    pRotation[i].Time = (MaxTime / (float)(KeyCount - 1)) * (float)i;

    // Retrieve initial rotation from transform matrix
    D3DXQuaternionRotationMatrix( &quatValue,
                                  &pFrame->TransformationMatrix );
```

At the start of the loop we calculate the time for the current keyframe we are adding for this node. To evenly space out our keyframes we divide the maximum time (600) by the total keycount - 1 (remember that keys are zero index based) to get a value of 31.57. This tells us we need a space of 31.57 ticks between each keyframe we add. As you can see, we then multiply this value by the current value of loop iteration variable ‘i’ (the current key we are processing) to generate the value for the current keyframe’s timestamp. For example, if we are adding the second keyframe, the timestamp is 1*31.57 = 31.57. Remember, the first keyframe would have been at position 0.0.

In the next step, we fetch the initial orientation of the frame’s parent-relative matrix as a quaternion representation using the D3DXQuaternionRotationMatrix function. Since rotation keyframes are stored as quaternions we can get the orientation of the frame in its reference pose as a quaternion and apply the rotation to that. We can then store the rotated quaternion in the keyframe to represent the orientation of the frame at this time. We know that later, the animation controller will fetch the quaternion and translation vectors for each keyframe and use them to rebuild the matrix for the frame. At this point, we need to do it the other way around. That is, we need to disassemble the matrix and store its orientation as a quaternion keyframe. In a moment, we will also apply a rotation to this quaternion so that it correctly represents the bone at the current time.

In the next three lines we will calculate how much rotation we wish to apply to the bone. It may look a little obfuscated, but just remember that all we are trying to do is find the correct rotation angle for this keyframe for the current node.

Although the next three lines of code are quite small, we will discuss each line one at a time so as we do not lose anybody along the way.

The first part of the calculation looks like this.

```
            // Calculate the angle of rotation
            float fInput = (((float)i / (float)(KeyCount - 1))
                           + fDelay) * (D3DX_PI * 2.0f);
```

When looking at the line above remember that 'i/(keycount-1)' is calculating the time of the current keyframe for this node in the 0.0 to 1.0 range. For example, we know there are 20 keyframes in total for this node, so if 'i' was currently 9, this would generate a parametric time value of 9/19 = 0.47 (approximate halfway through the 600 tick cycle). If we imagine this part in isolation, when calculated for every keyframe for this node, we are generating 20 parametric time values for each of the keyframes for the current bone. Obviously we will want each keyframe for this node to have a different rotation angle so that our branch sways backwards and forwards over time. It is intuitive then that the parametric time of the keyframe will need to play some part in its rotation angle.

We then add a Delay factor to this value. The Delay is in the range of –1.0 to 0.0 and will be closer to –1.0 the nearer to the ends of the branches the current bone we are processing is situated. Essentially, before the delay is added, we are calculating the actual time value parametrically for the current keyframe in the sequence for this bone. By adding the delay, we are effectively setting back the time (delaying) based on the position of the bone in the tree. While perhaps still not clear at this point, we have discussed how we wish the movement of bones nearer the top of the tree to seem more delayed than those near the bottom. So we are offsetting the parametric value by 0.0 to –1.0 based on the bone's position up the tree.

At this point that we have a possible range of values between –1.0 and 1.0 depending on the value of 'i' (the current keyframe being calculated for the node) and the delay (the iteration of the node in the tree). We will now map this –1 to +1 possible range of values into the –360 to +360 range by multiplying by 360 degrees ($\pi * 2$). Of course, we will work in radians, so the range of fInput after the above line will be [–6.28, +6.28].

Why would we want the position of the keyframe (with the appropriate delay added) mapped over the range of two circles (-360 to +360)? The reason is that we are going to use the shape of the cosine waveform to generate a scalar value which will be used to scale the wind strength value and the fMovement value we calculated earlier. At the top of the function, the fMovement value was calculated as a value between 0.0 and 1.0, based on the distance of the node from the base of the tree.

How does the cosine help us? To understand that, let us have a look at a graph of what the cosine function looks like if we plot its values over a large range of degrees (Figure 12.50).

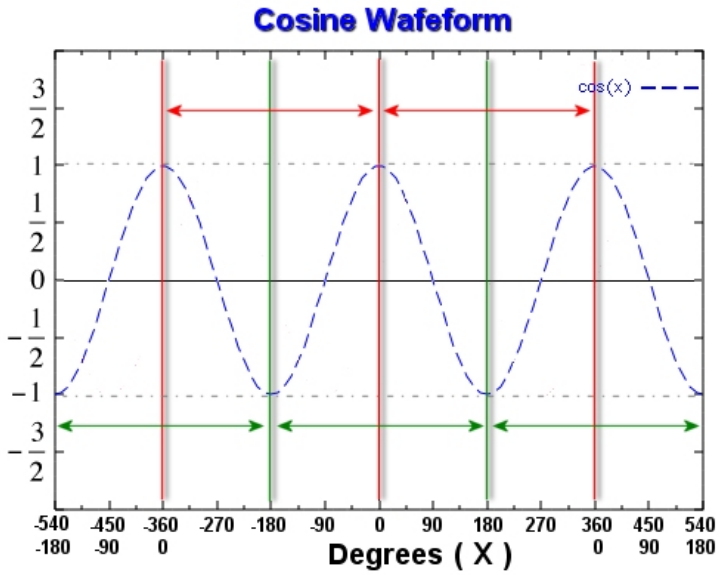

Figure 12.50

The output from the cosine function is between –1 and +1 over the range of a 360 degree circle. Furthermore, we can feed it values outside of this range and it will automatically wrap around to continue the shape of the waveform.

We can see for example that if we were to feed it values between 0 degrees (where the cosine is 1.0) to 180 degrees (where the cosine is –1) and then continue up to 360 degrees, the curve climbs back up again to a value of 1.0. This pattern is repeated as we increase or decrease values.

We can use the shape of this waveform to influence the angle by which we rotate a bone, such that the rotation pattern for the tree follows the cosine shape. For example, let us imagine that we are currently processing the first node in the tree, which would have a delay of zero. Also assume that there is a maximum of five keyframes for this frame. We know that in this case the value of fInput in the above calculation (for each keyframe) would be in the range [0, 360] degrees or [0, 6.28] radians. If we send each of these five values into the cosine function, we essentially plot five positions on the cosine waveform as shown in Figure 12.51.

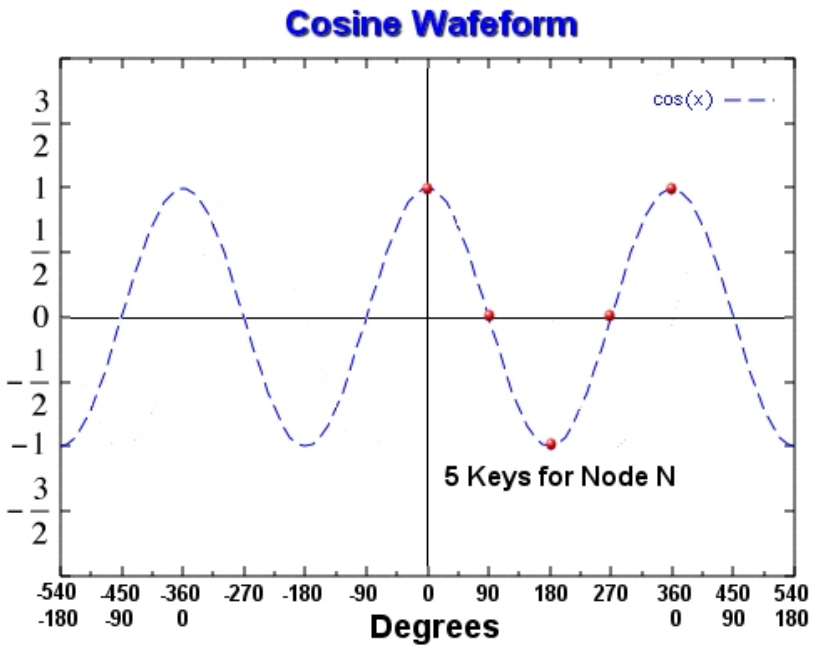

Figure 12.51

As you can see, the value assigned to each of the five nodes would be plotted on the graph and would return a value in the –1 to +1 range. This is not quite what we want at the moment, we want a value in the 0 to 1 range to act as our rotation scaling factor, but we will perform that mapping in a moment.

We see that the first node would have an input value of 0 and would therefore be assigned a height of 1 on the cosine graph. The next node would have an input of 90 degrees and would be assigned a height of 0. The third would have an input value of 180 and would be assigned a height of –180, and so on. By the time we get to the fifth node, the values start to repeat themselves.

If we imagine these values in the range of 0.0 to 1.0 (instead of –1 to +1) we can see that the result of the cosine function would return a scaling value, such that the first node would have full rotation applied, the second node half rotation, the third node no rotation, the fourth half rotation and the fifth full rotation, and the pattern continues. In other words, if we imagine these values being used to scale a rotation angle, we can see that the node would start at time zero in its fully rotated position, and over the five nodes, would rotate back to a rotation of zero and back out to a position of full rotation at the final node. Thus it is obvious why we are not happy with using the direct result from the cosine function -- it would be in the –1 to +1 range and we never want to invert the rotation angle, only scale it from zero to full rotation. The following code feeds in the input value into the cosine function and maps the result into the 0.0 to 1.0 range.

```
            float fCycle = (cosf( fInput ) + 1.0f) / 2.0f;
```

So using the above technique we essentially generate unique rotation values for each keyframe for a node. The pattern follows the shape of the cosine waveform to create a cyclical template for us to use (perfect for swaying our branches back and forth). But how did the initial fDelay value play a part? While the keyframes of the root node will always generate an fInput parameter in the range of 0-360

degrees (when fDelay=0), we must remember that the delay member can be in the range of 0.0 to –1.0. For the final node this would generate keyframes in the range of –360 to 0 degrees, which comes a whole cycle earlier on the cosine waveform. Therefore, what we are doing with this delay is moving all the nodes back from zero into the previous cosine cycle based on distance from the root node of the tree. For example, in Figure 12.52 we see how a node that is not the root node has had its cosine input value offset by the negative delay value. Note that the plotted points on the cosine graph lag behind the root node's plotted points.

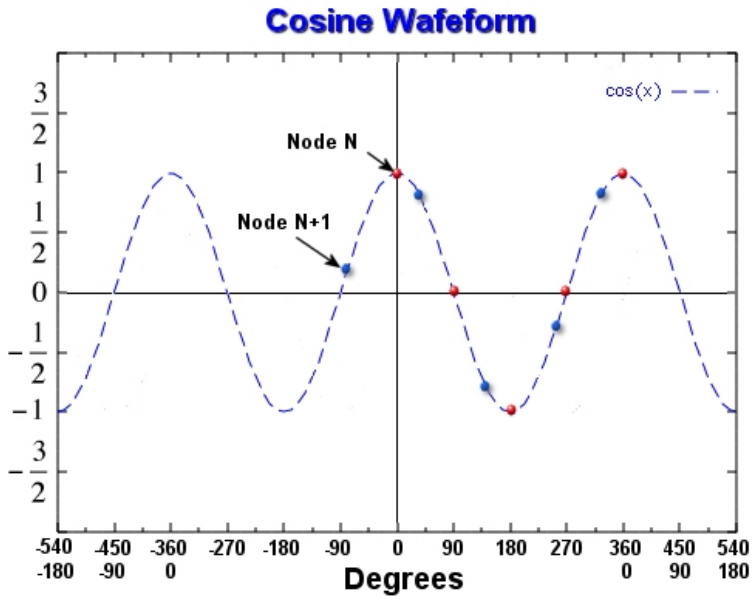

Figure 12.52

Hopefully you get the idea. The second node will have a different scaling value for each of its keyframes, but the pattern by which those angle scaling values transition from 0.0 to 1.0 will be the same. It will just be offset to some degree and mapped to a different interval of the waveform. Each node in the tree will still have the angle scaled by a complete 360 degree interval of the cosine waveform; it is just that for nodes further from the root, their initial start position on the waveform is shifted back to some position between 0 to –360.

We now have a local variable called fCycle which contains a value in the [0.0, 1.0] range. It describes how the rotation angle that we apply to this bone should be scaled based on the position of the node in the tree and the position of the keyframe we are currently processing in the keyframe list. What happens next?

Well, we know that the fWindStrength variable describes the strength of rotation in some way. Actually, we can think of this value as being the base rotation angle. For example, if a value of 30.0 was passed as the wind strength, this essentially describes a maximum rotation of 30 degrees at the branch end nodes (since the fCycle value we just calculated ranges from 0.0 to 1.0). So, we could just do fCycle * fWindStrength to get a rotation angle in the range of 0 to 30 degrees (in this example) that describes the rotation of the current keyframe. However, let us not forget that we calculated the fMovement value earlier (range = [0.0, 1.0]) which parametrically describes the distance from the

current node to the root node (0.0 = root node). As discussed, we definitely do not want the root nodes of the tree blowing back and forth nearly as much as the branch end nodes.

The fMovement value can be used to scale the wind strength angle (fMovement * fWindStrength) such that at the end nodes the rotation angle will be (1.0 * fWindStrength = Full Rotation) and at the root node it will be (0.0 * fWindStrength = No Rotation). So, we have the rotation angle for each node based on its position in the tree. This describes the maximum amount it will rotate through the sequence. We then must scale this by the fCycle value to account for the position of the current keyframe on the cosine waveform.

Angle = fCycle * (fMovement * fWindStrength)

If fWindStrength was 30 degrees and the node currently being processed was halfway up the tree from the root node, the fMovement value would be 0.5 and the maximum rotation angle for this node would be 15 degrees. Then, for each keyframe, fCycle would be a value in the range of 0.0 to 1.0, scaling the 15 degree rotation angle based on the position of the keyframe in the sequence and the delay for that node. Essentially, we generate a sequence where the bone will rotate between 0.0 and 15 degrees over its initial animation cycle and then rotate back to zero.

We are almost done, but not quite just yet. It will look unnatural if the wind is blowing our branches only to and from their default orientations. In real life, if we were to bend a branch and let it go, it would not neatly snap back to zero deviation, but likely overshoot the original center point (because of inertia) by some amount and oscillate. Perhaps it would bend 5 degrees in the negative direction before resetting itself. We will want to simulate the same thing at a simplistic level. Therefore, we will subtract 0.25 from the fCycle value to map it from [0.0, 1.0] to [–0.25, +0.75]. If a bone has a maximum rotation of 80 degrees, instead of rotating 80 degrees in one direction and then 80 degrees back to its starting position, it will actually rotate between –20 and +60 degrees. This means it will sway either side of its original starting position but with a ¾ bias in the wind direction. The final line that calculates the rotation angle is shown below.

```
  float fAngle = (fCycle - 0.25f) * (fMovement * fWindStrength);
```

We now have the angle, so our next task is to generate a rotation quaternion that represents a rotation angle equal the one we have just calculated for the wind rotation axis. To do this we will use the D3DXQuaternionRotationAxis function. We pass it an angle in radians and a rotation axis vector, and it will return a quaternion that represents the angle/axis rotation.

```
            // Rotate quaternion
            D3DXQuaternionRotationAxis( &quatRotate,
                                        &vecChildWindAxis,
                                        D3DXToRadian( fAngle ) );
```

Now all we have to do is multiply this rotation quaternion with the original quaternion we extracted from the frame matrix. This results in a quaternion that describes the orientation of the bone at this keyframe. Notice how we use the D3DXQuaternionMultiply function to rotate the original quaternion and then perform a quaternion conjugate on this quaternion to make it right handed. For some reason (as we discussed in Chapter 10), the D3DX animation system expects right handed quaternions even though

the D3DX quaternion functions generate left handed ones. The Conjugate function simply negates the axis of the quaternion so that it is pointing positive in the opposite direction (-x , -y , -z). We store the result of the conjugate directly in the keyframe's value. That completes the inner loop code and the process used to generate each of the 20 rotation keyframes for the current node.

```
            D3DXQuaternionMultiply( &quatValue, &quatValue, &quatRotate );

            // Conjugate quat into storage.
            D3DXQuaternionConjugate( &pRotation[i].Value, &quatValue );

        } // Next Key
```

As discussed earlier, we need to store at least two translation keyframes or GetSRT will interpolate between <0,0,0> and <0,0,0> (which will always return a result of <0,0,0>) and overwrite the true position of the frame when we call AdvanceTime. Therefore, for each node, we create two translation keyframes, one at the start of the timeline and one at the end, that both store the same frame position. This position is extracted from the translation vector of the frame matrix. The animation system will simply interpolate the translation value between the start position and the end position, which are the same, leaving the position of the frame constant.

```
       // Generate two bounding translation keys from original matrix
       pTranslation[0].Time  = 0.0f;
       pTranslation[0].Value = D3DXVECTOR3( pFrame->TransformationMatrix._41,
                                            pFrame->TransformationMatrix._42,
                                            pFrame->TransformationMatrix._43 );

        pTranslation[1].Time  = MaxTime;
        pTranslation[1].Value = D3DXVECTOR3( pFrame->TransformationMatrix._41,
                                             pFrame->TransformationMatrix._42,
                                             pFrame->TransformationMatrix._43 );

        // Register the animation keys
        hRet = pAnimSet->RegisterAnimationSRTKeys( pFrame->Name,
                                                   0,
                                                   KeyCount,
                                                   2,
                                                   NULL,
                                                   pRotation,
                                                   pTranslation, NULL );

    } // End if this is a bone node
```

In the above code, you can then see that after we have built the two translation vectors, we have all the information we need to build the animation for this frame in the hierarchy. We have an array of 20 rotation keyframes and 2 positional keyframes. We then register those keyframes with the animation set. Notice how we are careful to assign the name of the animation the same as the frame it is intended to animate. Remember, the animation controller uses this as its means for linking frames to animations. That ends the entire code block that is executed if the current node we are processing is a bone node. If not, all of the above code will be skipped and no animation will be created.

Finally, at the bottom of the function we see the common code that is executed for all nodes and not just bone nodes. It recurses along the sibling list if it exists and then down into the child list.

```
    // Build the nodes for child & sibling
    if ( pNode->Sibling )
    {
        hRet = BuildNodeAnimation( pNode->Sibling,
                                   vecWindAxis,
                                   fWindStrength,
                                   pAnimSet );

        if (FAILED(hRet)) return hRet;

    } // End if has sibling

    if ( pNode->Child )
    {
        hRet = BuildNodeAnimation(  pNode->Child,
                                    vecChildWindAxis,
                                    fWindStrength,
                                    pAnimSet );

        if (FAILED(hRet)) return hRet;

    } // End if has child

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

When the root iteration of this function (called from GenerateAnimation) returns, the animations for every bone in the hierarchy will have been constructed and added to the animation set. We saw earlier that when program flow returns, the GenerateAnimation function then registers the animation set with the controller and assigns it to an active mixer track.

That brings us to the end of our first tree discussion. Granted, there was quite a lot to take in, but it really did provide some very good insight into how skeletons and skins and animation all tie together.

12.6 Results

Lab Project 12.1 implements everything we have discussed so far in this chapter. Our tree class does not yet support adding leaves to the branches, but that is what the second part of this chapter will discuss. An example of a tree generated by Lab Project 12.1 is shown in Figure 12.53.

Figure 12.53

Admittedly, this does not look very impressive at the moment, but when we add leaves to this tree there will be a vast improvement. Since we will essentially just be adding new code to what we have already developed, it is recommended that, before reading any further, you examine the source code to Lab Project 12.1 and make sure that you understand the previous section of this textbook.

12.7 Adding Leaves to our Trees

Figure 12.54

For the remainder of this textbook we will discuss an upgrade to our tree system that allows us to significantly increase both detail and realism. Figure 12.54 shows a screenshot of a CTreeActor generated using the techniques we will discuss in this section. Looking at the image in Figure 12.54 it would seem as if our current code is a long way from creating trees of such detail. But as it happens, all that will be needed are a few (small) extra functions and a few minor tweaks to some existing code. After that, our CTreeActor class will be generating lush green trees as illustrated here.

The process of adding leaves to our tree will be one of adding a new branch node type to the virtual tree building process. Currently, a branch node can either be a BRANCH_BEGIN node which means it starts a new branch, a BRANCH_SEGMENT node which means it is just another segment in the branch, or a BRANCH_END node which means it ends the branch and represents the insertion of the tip vertex. We also know that any node in our virtual tree can have any of these nodes as its children. For example, we know that a given branch node might have three child nodes: a segment node that continues the branch and two branch start nodes that spawn new branches from that point. We will now introduce a new node type into our virtual tree referred to as a *frond node*.

Whenever a frond node is generated, we will insert special geometry into its parent branch mesh that will create leaves and the tiny little stalks that connect those leaves to their parent branch. You will see in a moment that while this might sound complicated, we represent a frond as two, two-sided quads criss-crossed and textured with a frond texture (see Figure 12.55).

> **Note:** The botanical meaning of the word "frond" is literally a leaf and its supporting structure (its stalk). So we will be adding not just leaves, but full fronds. That is, the leaves and the tiny twigs which connect them to their parent branch.

Figure 12.55

In Figure 12.55 we see the geometry that will be added to the branch mesh for a single frond node. The texture we apply to the intersecting quads is obviously going to be very important. In our application, we use a texture that contains not both the leaves and the stalk that will attach to the parent branch. This is important because when a frond node is randomly generated along a branch during the virtual tree building process, the position assigned to the frond node will be the same as the parent node that spawned it. That is, the base of the criss-crossed polygons shown in Figure 12.55 will actually begin

inside the branch mesh (at the center of the parent branch segment). As such, the stalk of the frond will be seen sticking out of the parent branch (like little branches flourishing into leafy tips).

With regards to the virtual tree building process, frond nodes are like normal branch nodes. They have a position and a direction vector which is deviated from the parent just like normal branch nodes. There is only one real difference between a frond node and a normal branch node; a frond node will never have any child nodes of its own. We can think of them as being child branches that are always one node in length. In this way, we can think of a frond node as being a little like a child branch being spawned in the traditional sense, where its first branch node is a branch end node. Frond nodes can be spawned from any other branch node, and although it is possible to have any given branch node spawn multiple frond nodes, in our implementation we found that this was not necessary. By giving each node a chance to create a single child front node we get trees with plenty of foliage. When the frond geometry created at one branch node overlaps the frond geometry created at its neighboring branch node we get a jumbled mix of leaves that can look very full and bushy.

In Figure 12.55 we can see that by adding four quads for a given frond node (remember that both the intersecting quads must be two-faced, which equates to four quads in total) we are able to simulate a lot of detail with a very low polygon count. Of course, much depends on how detailed the frond texture is. If the texture contained a single leaf, many more frond nodes would have to be created to build up a busy looking tree (and probably would not look very good anyway). The texture image we used in our demonstration actually contains multiple leaves and their stalks, connected by a central thin branch. The frond geometry created in Figure 12.55, if added to our tree and rendered in the way illustrated, would look quite terrible because we can clearly see the black background of the texture. This makes it a little too obvious that we are just mapping textures to quads and randomly scattering them through the tree.

Figure 12.56

Not surprisingly, our frond texture has an alpha channel so that we can enable alpha testing in the pipeline and filter out the black background pixels. Only the branch and leaf pixels will be rendered to the frame buffer. This allows other branches and frond constructs to be seen through the gaps in those leaves. We will set the pipeline alpha testing mechanism to reject any pixels from the texture that are below a certain alpha value. This will stop nearly all the background pixels being rendered.

The problem with just performing this test however is that if you study the alpha channels for such textures, they usually blend from transparent to opaque around the outside of the leaves. Therefore, even with alpha testing enabled, we can still see a dark silhouette around the leaf boundaries where the pixels are slightly darker and have alpha values that are just above the rejection value set as the alpha testing reference. If we enable alpha blending, we can remedy this using the SRC_ALPHA and INV_SRCALPH blend modes from the source and destination blend modes respectively. This will cause what was once a dark silhouette around the main image to blend with the contents of the frame buffer. With alpha blending and alpha testing enabled, the boxes around the frond texture image are removed, the quads are no longer visible, and we

end up with quite organic looking foliage (see Figure 12.56). This is a very common technique in real-time games and it is used to produce all manner of foliage, not just fronds.

12.7.1 The Frond Node

Our virtual tree building function (BuildBranchNodes) will have to be modified so that at any given branch node, it also has the ability to spawn child frond nodes. It is important to realize at this point that the frond node is not really any different from a normal branch node. The node's direction and right vector will still be deviated from its parent and the direction vector of the frond node will be used in the mesh building process to construct the direction of the frond construct. Just as in the normal branch building process, where the branch node position and direction vector describe a plane on which a ring of vertices will be placed, the frond node normal and position describe the orientation of a plane on which the base of the frond quads will be mounted (Figure 12.57).

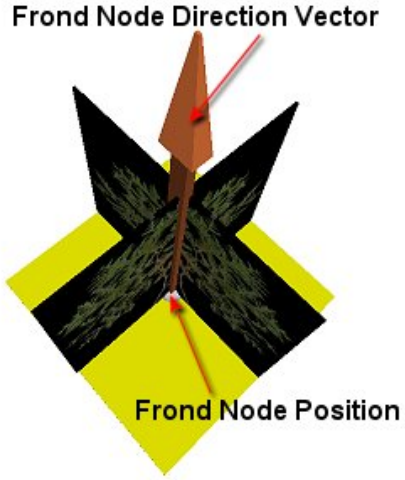

Figure 12.57

In Figure 12.57 the large brown arrow describes the direction of the node which was randomly deviated during node creation. The yellow slab represents the plane described by the node position and the direction vector. If this was a normal branch node, this the plane on which the ring of vertices would be placed. However, because this is a frond node, during the mesh building process we will build the two intersecting two-sided quads where the bottom vertices of each quad lay on the plane and the top vertices of each quad are offset from the plane in the direction of the node plane normal. That way, whichever way the plane is facing, the quads will always have their bottom vertices attached to the plane and the fronds will always point in the direction described by the node direction vector. As you have no doubt gathered by now, the only real difference between a normal branch node and a frond branch node is how each is treated during the mesh creation phase. The building of the virtual tree of branch node structures will require very few modifications. All it needs is the ability to randomly create frond nodes, just like it creates any other child node.

So that we are clear on how these nodes will be placed and how the geometry for these frond nodes will be generated, before we discuss the code let us just look at an example of how multiple frond nodes may be created to provide a nice combined visual effect.

In Figure 12.58 we see a tree that has a branch that contains a single frond node between its second and third branch segment. The position of the frond node will be inherited from its parent. Therefore, we can see that the frond node is the child of the branch node that forms the end of segment two and the beginning of segment three in the parent branch. We can also see that at the exact same spot the frond node is spawned, a new child branch is also spawned.

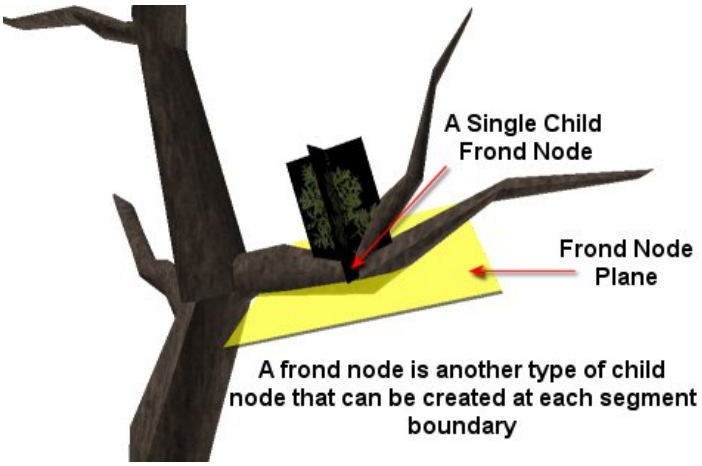

Figure 12.58

As discussed, the frond node is assigned an arbitrary direction just like any other node. The direction vector of this particular frond node, along with its position, would describe the orientation of the yellow plane in the diagram. Notice how the orientation of the frond quads is such that they are aligned to this plane. It is like the plane forms the base on which the quads are mounted and as such, the orientation of this plane determines the orientation of the frond construct. Therefore, we can think of the direction vector of the frond node as not just describing the normal to this plane, but also describing the orientation of the central spine of the frond construct where the quads intersect one another. So generating a frond node is identical to generating any other node during the virtual tree building process. We just generate a new node of type frond and assign it the direction and right vectors as normal.

The position and frequency at which frond nodes are generated will be controlled by some additional members in our TreeGrowthProperties structure. Of course, we will generally want to spawn frond nodes much more frequently than branch nodes so that we get satisfactorily dense foliage. We will often wish to spawn many child frond nodes from each parent. For example, Figure 12.59 shows the same parent node in the branch now spawning multiple frond nodes. Notice how each frond node (just like a branch start node) is randomly deviated so that we have multiple frond constructs originating from the same position in the branch but having arbitrary directions. In this example, the branch node has spawned four frond nodes as children, and also exists in a sibling list with a branch start node.

Figure 12.59

When we couple the fact that every node that meets the frond growth requirements (specified in the TreeGrowthProperties structure) can spawn multiple child frond nodes with the fact that most branch nodes will spawn child frond nodes, we have a situation where neighboring branch nodes spawn multiple fronds which all collide and overlap with each other making for a very busy looking tree. Figure 12.60 demonstrates this concept by showing just two branch nodes in close proximity that have each spawned multiple frond nodes. Even in this example, the tree is starting to look pretty busy. Imagine the same number of frond nodes being generated at most branch nodes and we can image how full our foliage will look. Because fronds from neighboring branch nodes collide and overlap it actually adds to the chaotic realism of the tree.

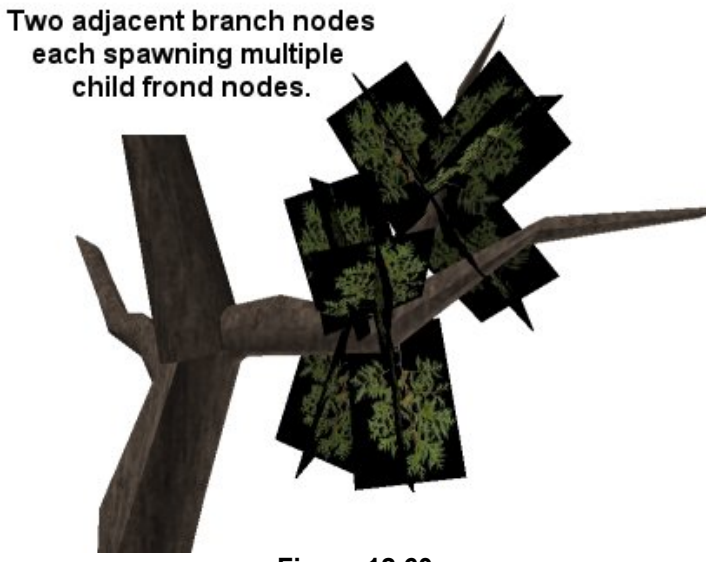

Figure 12.60

If we take the example shown in Figure 12.60 and add alpha testing and alpha blending, we can see that the results are starting to look quite pleasing even in this example where only two branch nodes are spawning fronds. The alpha tested/blended result is shown in Figure 12.61.

Figure 12.61

> **Note:** In our demo implementation we only generate one child frond node for any given parent. That is, a branch node will either have zero or one child frond nodes and never any more. However, this is certainly a behavior you can change to generate trees with more dense foliage. At the moment we are talking generically about the technique.

Now that we know exactly where frond nodes should exist and we understand the geometry they use, we can discuss the additions to the CTreeActor class that will need to be made to add frond support to the code developed in Lab Project 12.1. As we will, for the most part, simply be discussing additions to pre-existing functions, it is highly recommended that while reading this next section you open up the source

code to lab project 12.2 so that you can follow along with the code changes made in this section of the textbook. In most cases, we will not be showing all of the function code again, but only discussing the changes that have been made.

12.7.2 Updating the Virtual Tree Generation Process

As we know from the coverage of Lab Project 12.1, the first phase of the tree generation process is the construction of the virtual tree (the tree of BranchNode structures). Phase one is activated from the GenerateTree method (called by the application) via a call to the function GenerateBranches. You will recall that this function simply created the root branch node (or multiple root nodes if applicable) and then called the BuildBranchNodes function to kick off the recursive process for each root. The BuildBranchNodes function recursively called itself, adding branch nodes to the virtual tree hierarchy until it had been completely built.

For any given instance of the BuildBranchNodes method there was a simple task. The underlying question was, using the probabilities specified in the TreeGrowthProperties structure, should the current node being processed spawn a child node that continues the branch or an end node that ends it? Apart from this, it also decided how many BRANCH_BEGIN nodes should be generated as children of the current node. Recall that a BRANCH_BEGIN node signifies the beginning of a new child branch spawning from the parent.

This function will have to be modified so that it now has the option to randomly generate child frond nodes. Previously the strategy was simply:

1. Does this branch continue?
   Yes: Create child BRANCH_SEGMENT node and recur into Child Node
   No : Create child BRANCH_END node and recur into Child Node

2. Do we wish to create any child branches?
   Yes: Decide how many and create each child BRANCH_BEGIN node and recur into each
   No : Do nothing and just return

Of course, any given instance of the function returned if it had stepped into an end node. This logic will not change, but it will be supplemented. As you can see, the first step was to determine what type of node the next node in the current branch will be and to process it. The second step was to determine if child branches should also be started at this node and have the relevant child nodes created.

Now, after recursively processing the node that continues the branch and any child branch start nodes, we will add a third section that decides whether we would like to add a child frond node. Currently, each branch node can be one of three types: BRANCH_BEGIN, BRANCH_SEGMENT, or BRANCH_END. Now we will add a fourth type to the BranchNodeType enumeration called BRANCH_FROND. The updated enumeration, which is part of the CTreeActor namespace, is shown below.

Excerpt from CTreeActor.h

```
enum BranchNodeType {   BRANCH_BEGIN      = 1,
                        BRANCH_SEGMENT    = 2,
                        BRANCH_END        = 3,
                        BRANCH_FROND      = 4 };
```

Whether we create a frond node at any given parent node will once again be a random decision within limits given by the probabilities set in the TreeGrowthProperties structure. We will add five members to this structure which will be used by the virtual tree generation process to decide if a frond node should be created at a given node in the tree. Below we see the five new members of the TreeGrowthProperties structure (as defined in CTreeActor.h) followed by a description of each member and how it is used by the BuildBranchNodes method.

TreeGrowthProperties Structure (additions)

```
bool        Include_Fronds;             // Include fronds in the build
USHORT      Min_Frond_Create_Iteration; // Min iteration fronds can be created.
float       Frond_Create_Chance;        // Chance frond will be created at a node
D3DXVECTOR3 Frond_Min_Size;             // The smallest size a frond can be
D3DXVECTOR3 Frond_Max_Size;             // The largest size a frond can be
```

bool Include_Fronds

This boolean is used as a frond switch. Setting it to false will disable the generation of frond nodes and we will get trees identical to the ones from Lab Project 12.1. Setting this to true (default) will enable frond generation.

USHORT Min_Frond_Create_Iteration

We want to be able to control the node iteration at which frond nodes are created. The concept of a node's iteration is not new to us; it describes depth in the hierarchy (the number of branch segments that would need to be traversed from the root of the tree to reach that node). This value describes how many nodes deep in the hierarchy we must be before frond nodes are created. We do not usually want this to be set to a low number (or zero) or we will see frond nodes at the root of the tree. Usually, we want the fronds to start only when we get a certain way up the tree and out along the branches. The default value is 6. So fronds will not be generated at a given node while its depth in the hierarchy is smaller than 6 levels below than the root.

float Frond_Create_Chance

This property which (range = [0.0, 100.0] describes the probability that a frond node will be created as a child of the current node being processed (assuming the hierarchy depth test passes). For any given non-frond node that we are processing we will generate a random value between zero and one hundred. If the value is less than this probability, a new child frond node will be generated. Higher values for this property create more fronds and bushier trees.

D3DXVECTOR3 Frond_Min_Size
D3DXVECTOR3 Frond_Max_Size

We will usually want our fronds to become smaller in size as they progress up the tree. These two vectors describe the minimum and maximum sizes that a frond can fall between. These vectors actually describe the half-lengths of the intersecting quads (x,y) that will be generated. The z components of these vectors describe the height range for the quads (i.e., the height with respect to the node plane they

are mounted on). Larger z values will essentially create longer fronds. We will see how these values are used in the branch mesh creation phase (phase 2).

Although these properties (along with all the other tree growth properties) can be set by the application using the CTreeActor::SetGrowthProperties method, default values are specified in the constructor, as shown below.

Excerpt from CTreeActor.cpp : CTreeActor::constructor

```
// Setup frond defaults
m_Properties.Include_Fronds             = true;
m_Properties.Min_Frond_Create_Iteration = 6;
m_Properties.Frond_Create_Chance        = 25.0;
m_Properties.Frond_Min_Size             = D3DXVECTOR3( 7.5f, 7.5f, 11.25f );
m_Properties.Frond_Max_Size             = D3DXVECTOR3( 10.5f, 10.5f, 15.75f );
```

Let us now take a look at the code that has been added to the end of the BuildBranchNodes function to allow for the random generation of a child frond node.

CTreeActor::BuildBranchNodes - Updated

It is advisable to load up the source code to the BuildBranchNodes function in Lab Project 12.2. This function was covered in a lot of detail in the first section of this chapter and should be fully understood. We are now going to add a new conditional code block to the bottom of this function to create a new child BRANCH_FROND node. We will show only the new code.

```
…
…
…
} // End create child branch nodes


// Should a frond be created as a child of this node?
if ( m_Properties.Include_Fronds &&
     Iteration >= m_Properties.Min_Frond_Create_Iteration
     && (bEnd || ChanceResult( m_Properties.Frond_Create_Chance )) )
    {
```

In the above code we see that we only enter this new code block if the iteration value of the current node being processed is larger or equal to the Min_Frond_Create_Iteration growth property and if the boolean growth property bInclude_Fronds has been set to true. If these conditions are true and the ChanceResult function returns true (notice we feed in the Frond_Create_Chance growth property as the parameter) or if we have just created a child BRANCH_END node, we will create a child frond node. (We will always generate a frond node at a branch end node provided its iteration value is larger than Min_Frond_Create_Iteration).

If we get into this code block then we wish to create a new frond node. We allocate a new branch node and set its properties much like we do any other node.

```
        // If this node was the end of a branch, insert a frond node
        BranchNode * pFrondNode = new BranchNode;
        if ( !pFrondNode ) return;

        // Store node details
        pFrondNode->UID          = BranchUID++;
        pFrondNode->Parent       = pNode;
        pFrondNode->Dimensions   = pNode->Dimensions;
        pFrondNode->Position     = pNode->Position;
        pFrondNode->Direction    = pNode->Direction;
        pFrondNode->Right        = pNode->Right;
        pFrondNode->BranchSegment= 0;
        pFrondNode->Iteration    = (USHORT)Iteration + 1;
        pFrondNode->Type         = BRANCH_FROND;
```

Notice how the position and dimensions are inherited from the parent node and it is assigned its own unique ID like all other branch nodes. This time however, the Type member of the BranchNode structure is set to BRANCH_FROND. We inherit the direction and right vectors from the parent and these will be deviated in the next line of code using the same deviation angle ranges as a normal BRANCH_BEGIN node.

```
        // Deviate the node
        DeviateNode( pFrondNode,
                     m_Properties.Split_Deviation_Min_Cone,
                     m_Properties.Split_Deviation_Max_Cone,
                     m_Properties.Split_Deviation_Rotate );
```

At this point we have created a new frond node and have it located at the position of the parent node, pointing in an arbitrary direction (just like the start of a new branch). However, the frond node is not yet attached to the parent node's child list.

We need to make sure we add it to the right place in the child list if the parent node already has child nodes. It is important when building the skinning information that if the parent node contains a branch end node child, that the frond node be placed before it in the list. The vertices of the frond will be added as influences of the previous bone that was created back from it along the branch, so we wish to add these vertices before hitting the branch end node, where we add the single tip vertex and abort any further processing of the branch. The next section of code makes sure it is added to the child list prior to any BRANCH_END node that may exist there.

```
        // Link the node
        if ( !pNode->Child )
        {
            // No child already exists, just store
            pNode->Child = pFrondNode;

        } // End if no child node
```

As the above code shows, if the parent node has no other children, then we just assign its child pointer to point at the new frond node. Our new frond node will be the only node in the child list. Otherwise, the next code block is executed.

The following code sets up a while loop to traverse the child list and breaks from the loop as soon as a BRANCH_END node is found. Because with every iteration of the loop it stores the current child in its pPrev pointer before stepping into the next sibling, if the loop breaks because a BRANCH_END node is found, the pPrev pointer will point at the sibling node that exists prior to the end node in the list (the node to which we wish to add our new frond node as a sibling). That is, we will sandwich our new frond node between the branch end node and the node that existed prior to it in the sibling list.

```
        else
        {
            BranchNode * pTemp = pNode->Child, *pPrev = NULL;
            while ( pTemp )
            {
                // Have we found the end of the branch?
                if ( pTemp->Type == BRANCH_END ) break;
                pPrev = pTemp;
                pTemp = pTemp->Sibling;

            } // Next sibling

            if ( !pPrev )
            {
                // The BRANCH_END is right at the start.
                pFrondNode->Sibling = pNode->Child;
                pNode->Child = pFrondNode;

            } // End if no previous item
            else
            {
                // Attach to the 'previous' item.
                pFrondNode->Sibling = pPrev->Sibling;
                pPrev->Sibling = pFrondNode;

            } // End if previous item exists

        } // End if child node

    } // End if create frond

} // End function
```

And that represents all the code changes that had to be made in BuildBranchNode.

What is important to notice in the above code is that unlike normal branch node creation, where after it is created we recur into that node, for frond nodes we do not do this. We can think of a frond node as being a single node branch. We never recur into a frond node or add other child nodes to it.

12.7.3 Setting the Frond Texture and Material

In our previous incarnation of CTreeActor we had the ability to store the texture and material that would be used to render the branches of the tree. We exposed a method to allow the application to set these properties prior to the call to the GenerateTree function. We will now need to add two new members to CTreeActor that will contain the texture filename and material that will be used for the fronds. An accompanying member function will allow the application to set these properties prior to tree generation.

Excerpt from CTreeActor (Additional Member Variables)

```
D3DMATERIAL9 m_FrondMaterial;    // The material to apply to the fronds.
LPTSTR       m_strFrondTexture;  // The texture to apply to the fronds.
```

CTreeActor – SetFrondMaterial

This function accepts two parameters: a string containing the name of the image file we would like to use as the texture for the fronds and a pointer to a D3DMATERIAL9 structure. The texture and material are copied into the class member variables.

```
void CTreeActor::SetFrondMaterial( LPCTSTR strTexture, D3DMATERIAL9 * pMaterial )
{
    // Free any previous texture name
    if ( m_strFrondTexture ) free( m_strFrondTexture );
    m_strFrondTexture = NULL;

    // Store the material
    if ( pMaterial ) m_FrondMaterial = *pMaterial;

    // Duplicate the texture filename if any
    if ( strTexture ) m_strFrondTexture = _tcsdup( strTexture );
}
```

In the next section we will discuss the upgrades to the CTreeActor::BuildNode method. We discovered in the first section of this textbook that this method implements the second phase where the actor's hierarchy and branch skins are constructed. We will only examine the portions containing the updated code, so it is recommended that you open Lab Project 12.2 and follow along in the next section using the actual source code.

Updating CTreeActor::BuildNode

This function is called once by the BuildFrameHierarchy function and is passed the root node of the virtual tree. It steps through the hierarchy recursively from the root node visiting each branch node in the tree. For a given node, this function would add a frame to the actor's hierarchy if it was determined that

the current node should generate a bone. If the node is a BRANCH_BEGIN node, a new CTriMesh would be created and the branch would be traversed to add the vertex and index data. For any non-branch node that is not a BRANCH_BEGIN node the passed mesh is the mesh that is currently being built for the current branch being processed (i.e., the mesh that the vertices generated at this node should be added to).

In the first section of the code shown below you can see that we have snipped out nearly all the logic that generates the new frame if this is a bone node. A bone is never generated for a BRANCH_END node and likewise, if it is a BRANCH_FROND node, a bone will also *not* be created. Therefore, the code inside this conditional is completely unchanged (the conditional code never gets executed if the current node is a frond node).

We have removed all the code that generates the new frame and attaches it to the hierarchy as well as the code that creates the new CTriMesh if the current node is a BRANCH_BEGIN (bones are always added to the start of a branch). What we have left in place as a reminder at the bottom of the code block is the call to the AddBranchSegment method which was responsible for adding the ring of vertices and their indices to the mesh.

```
HRESULT CTreeActor::BuildNode( BranchNode * pNode,
                               D3DXFRAME * pParent,
                               CTriMesh * pMesh,
                               const D3DXMATRIX & mtxCombined,
                               ID3DXAllocateHierarchy * pAllocate )
{
    HRESULT     hRet;
    CTriMesh   *pNewMesh  = NULL, *pChildMesh  = NULL;
    LPD3DXFRAME pNewFrame = NULL,  pChildFrame = NULL;
    D3DXMATRIX  mtxBranch, mtxInverse, mtxChild;
    D3DXVECTOR3 vecX, vecY, vecZ;
    TCHAR       strName[1024];

    // What type of node is this
    bool bIgnoreNodeForBone = (pNode->Type == BRANCH_END ||
                               pNode->Type == BRANCH_FROND);

    if ( pNode->Type == BRANCH_BEGIN ||
        ((pNode->BranchSegment % m_Properties.Bone_Resolution) == 0 &&
         !bIgnoreNodeForBone) )
    {
        …
        …
        …
        …  --- snip ---   ( Create Frame and Setup Parameters to pass to child)
        …
        …
        …

        // Add the ring for this segment in this frame's combined space
        hRet = AddBranchSegment( pNode, pChildMesh );
        if ( FAILED(hRet) ) { if (pNewMesh) delete pNewMesh; return hRet; }

    } // End if adding a new frame
```

The next section of code was executed if the current node is not a BRANCH_BEGIN node and is not a node that should have a bone created for it. You will recall that originally this just meant calling the AddBranchSegment function to add a ring of vertices to the current mesh being built (the one passed into the function from the parent). Now we have some conditional logic. If the current node is a BRANCH_FROND node then we do not call the AddBranchSegment function since we do not wish a ring of vertices to be inserted on the frond node plane. Instead, we wish to insert vertices into the mesh that will build the intersecting quads for the frond. Therefore, we call a new method called AddBranchFrond to add the frond geometry to the current mesh. This function will be discussed in a moment.

```
    else
    {
        // Store which bone we're assigned to in the node
        pNode->pBone = pParent;

         // Is this a frond?
        if ( pNode->Type != BRANCH_FROND )
        {
            // Add the ring for this segment in this frame's combined space
            hRet = AddBranchSegment( pNode, pMesh );
            if ( FAILED(hRet) ) { return hRet; }

        } // End if not frond
        else
        {
            // Add the frond data in this frame's combined space
            hRet = AddBranchFrond( pNode, pMesh );
            if ( FAILED(hRet) ) { return hRet; }

        } // End if frond

        // Since no new frame is generated here, the child will receive
        // the same frame, mesh and matrices that we were passed.
        pChildFrame = pParent;
        pChildMesh  = pMesh;
        mtxChild    = mtxCombined;

    } // End if not creating a new frame
```

Because we only add fronds when we are not processing a bone node you might conclude that there would be gaps in the foliage whenever a branch node is a bone node. However, we must remember that this function also visits the sibling list of the bone node, so fronds can still exist at the same position as a bone node and can exist in the same list of siblings as any node type. At this point we are outside any conditional code block and back in the common flow of the function.

The next section of code steps into the child and sibling lists (this is not new code).

```
    // Build the nodes for child & sibling
    if ( pNode->Sibling )
    {
        hRet = BuildNode( pNode->Sibling,
                          pParent,
```

```
                        pMesh,
                        mtxCombined,
                        pAllocate );

        if (FAILED(hRet)) { if ( pNewMesh ) delete pNewMesh; return hRet; }

    } // End if has sibling

    if ( pNode->Child )
    {
        hRet = BuildNode( pNode->Child,
                        pChildFrame,
                        pChildMesh,
                        mtxChild,
                        pAllocate );

        if (FAILED(hRet)) { if ( pNewMesh ) delete pNewMesh; return hRet; }

    } // End if has child
```

The final section of code is executed at the bottom of the function when the current node being processed is a BRANCH_BEGIN node. Remember, because the child list is traversed before we get to this point in the function, if this is a BRANCH_BEGIN node, all the vertices in that child branch will have been added to the mesh at this point. What we have to do in this next code block is build the CTriMesh's underlying ID3DXMesh, build its ID3DXSkinInfo object, and pass this information into the CAllocateHierarchy::CreateMeshContainer function for skinned mesh creation. Virtually all of this code is unchanged, but we will comment after the first altered portion.

```
    if ( pNode->Type == BRANCH_BEGIN )
    {
        D3DXMATERIAL        Materials[2];
        D3DXMESHDATA        MeshData;
        D3DXMESHCONTAINER * pNewContainer       = NULL;
        DWORD             * pAdjacency          = NULL;
        LPD3DXBUFFER        pAdjacencyBuffer    = NULL;
        LPD3DXSKININFO      pSkinInfo           = NULL;
        ULONG               MaterialCount = ( m_Properties.Include_Fronds ) ? 2 : 1;
```

Notice the new line added to the bottom of the variable declarations. We will have to pass into the 'CreateMeshContainer' function the number of materials/subset in the mesh. If fronds are enabled we will have two subsets/materials in the mesh, so we perform this calculation here. The first subset will contain the branch faces and the second the fronds.

```
        // Generate mesh containers name
        _stprintf( strName, _T("Mesh_%i"), pNode->UID );

        // Generate the skin info for this branch
        hRet = BuildSkinInfo( pNode, pNewMesh, &pSkinInfo );

        if ( FAILED(hRet) ) { delete pNewMesh; return hRet; }

        // Signal that CTriMesh should now build the mesh in software.
```

```
        pNewMesh->BuildMesh( D3DXMESH_MANAGED, m_pD3DDevice );

        // Build the mesh data structure
        ZeroMemory( &MeshData, sizeof(D3DXMESHDATA) );
        MeshData.Type  = D3DXMESHTYPE_MESH;

        // Store a reference to our build mesh.
        // Note: This will call AddRef on the mesh itself.
        MeshData.pMesh = pNewMesh->GetMesh();

        // Build material data for this tree
        Materials[0].pTextureFilename = m_strTexture;
        Materials[0].MatD3D           = m_Material;
        Materials[1].pTextureFilename = m_strFrondTexture;
        Materials[1].MatD3D           = m_FrondMaterial;
```

The call to the BuildSkinInfo function is not changed although the contents of this function have been modified slightly, as we will see in a moment. The BuildSkinInfo function has also been upgraded so that any frond vertices also get influenced by the bones of the tree. The two new lines of code in the above code snippet are the bolded ones at the very bottom.

You will recall that when we pass our new branch mesh into the CreateMeshContainer function we must also pass in an array of materials (texture and material combinations). Originally, we just passed in a single structure since we only had one texture and material used by the entire tree. Now we have two because we also have a texture and material that will be used for the fronds (the second subset). So as you can see, we store both of these in a two element D3DXMATERIAL array before handing it off to CreateMeshContainer. Notice that we pass in the material count we calculated earlier so that CreateMeshContainer knows how many materials we are passing in the array.

```
        // Retrieve adjacency information
        pNewMesh->GenerateAdjacency( );
        pAdjacencyBuffer = pNewMesh->GetAdjacencyBuffer();
        pAdjacency       = (DWORD*)pAdjacencyBuffer->GetBufferPointer();

        // Create the new mesh container
        hRet = pAllocate->CreateMeshContainer( strName,
                                               &MeshData,
                                               Materials,
                                               NULL,
                                               MaterialCount,
                                               pAdjacency,
                                               pSkinInfo,
                                               &pNewContainer );

        // Release adjacency buffer
        pAdjacencyBuffer->Release();

        // Release the mesh we referenced
        MeshData.pMesh->Release();

        // Release the skin info
        if (pSkinInfo) pSkinInfo->Release();
```

```
        // Destroy our temporary child mesh
        delete pNewMesh;

        // If the mesh container creation failed, bail!
        if ( FAILED(hRet) ) return hRet;
```

The final change to this function requires some explanation. We will need to override the DrawActor and DrawActorSubset methods of the base class in CTreeActor so that we can enable alpha testing/blending when rendering the frond subset (subset 1). However, if our actor is in managed mode, the CreateMeshContainer function will have remapped the face attribute IDs to use the global IDs issued by the CScene class during the creation of the skin. We currently have no way of knowing what the frond subset ID will be after the CreateMeshContainer function has altered its attribute buffer. Furthermore, we cannot even lock the attribute buffer and try and determine this because the mesh may have also been attribute sorted (optimized) so we cannot even be sure that the first subset ID in the attribute buffer is the ID of the non-frond subset.

We decided to work around this problem by making a very small tweak to our CAllocateHierarchy::CreateMeshContainer function. In the next Module in this series we will examine a more comprehensive solution to this problem, but for now, our current strategy will be a lot easier to deal with.

You will recall that during the attribute remapping of a non-managed mesh we would build a temporary remap array describing what the original attribute IDs were mapped to. We then used this array to remap the attribute buffer and then the array was discarded. We will no longer discard it. Instead we store the attribute remap array in a new CAllocateHierarchy member variable so that it can be accessed. We added a small function to the CAllocateHierarchy class called GetAttributeRemap. This function will return the new ID in the remap array for an original subset ID passed in. In the following code you can see us using this new function to retrieve the new subset ID for the frond subset (originally subset 1) and storing it in a new CTreeActor member variable called m_nFrondAttribute. Our overridden DrawActorSubset subset function can then access this attribute ID and only enable the alpha blending/testing render states if the frond subset is the one that has been requested to be rendered.

```
        // Store the final attribute ID (as it was remapped) of the frond data
        if ( m_Properties.Include_Fronds )
        m_nFrondAttribute = ((CAllocateHierarchy*)pAllocate)->GetAttributeRemap( 1 );

        // Store the new mesh container in the frame
        pNewFrame->pMeshContainer = pNewContainer;

    } // End if beginning of branch

    // Success!!
    return D3D_OK;

}
```

If you followed along in the last section with your source code project open you will have seen that the changes to the code are very small indeed. However, there is now a new function called from the

BuildNode function which adds the vertex data to the mesh for a frond node. This function is called AddBranchFrond and is a brand new function whose code we will cover next.

CTreeActor - AddBranchFrond

The AddBranchFrond function is called by BuildNode whenever a frond node is encountered during phase two of the tree building process. Just as the AddBranchSegment function is called to add a ring of vertices on the node plane of a normal branch node, the AddBranchFrond function is called to add the frond vertices. This function will create two intersecting quads mounted on the node plane and aligned with the node direction vector (see Figure 12.62).

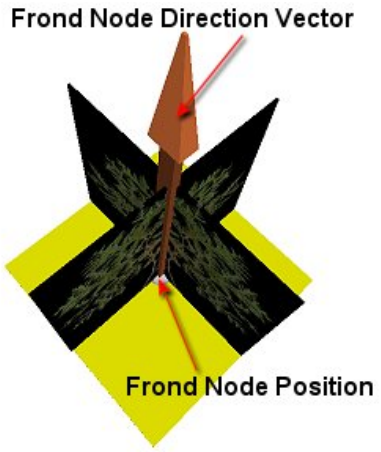

Figure 12.62

Just as we did in the AddBranchSegment function, we must transform the node position and the node plane's direction, right, and ortho vectors from tree space into branch space. This is so the vertices we generate will be relative to the beginning of the branch (the mesh in which they are contained) and not the root of the entire tree. The process we use to perform this back transformation is the same. The function is passed the frond node that needs to have its vertices added and a pointer to the CTriMesh to which the vertices should be added (the current branch mesh being built). It first executes a while loop that starts at the current node and steps back through the parent list until the BRANCH_BEGIN node is encountered. This will be the first node in the branch and the space in which we wish to transform our frond node. That is, we wish to define our frond vertices in a space where the BRANCH_BEGIN node's position is located at (0,0,0) in the coordinate system and its right, ortho and direction vectors are aligned with the X,Y, and Z axes of that system, respectively.

```
HRESULT CTreeActor::AddBranchFrond( BranchNode * pNode, CTriMesh * pMesh )
{
    USHORT      Index;
    D3DXMATRIX  mtxInverse, mtxRot, mtxBranch;
    D3DXVECTOR3 vecPos, vecAxis, vecRight, vecOrtho, vecNormal, vecVertexPos;
    D3DXVECTOR3 vecX, vecY, vecZ;
    float       fScale, fX, fY, fZ;

    // Back track until we find the beginning node for this branch
    BranchNode * pStartNode = pNode;
    while ( pStartNode->Type != BRANCH_BEGIN && pStartNode )
            pStartNode = pStartNode->Parent;

    if ( !pStartNode ) return D3DERR_INVALIDCALL;
```

At this point we now have a pointer to the BRANCH_BEGIN node, so let us extract its direction vector (it local Z axis) and its right vector (its local X axis) and perform a cross product to generate the third axis in its local system (its local Y axis). Using these three vectors we then build a transformation matrix for the branch start node and take its inverse. This gives us a matrix that will transform any position or vector into branch space.

```
    // Store / generate the vectors used to build the branch matrix
    vecX = pStartNode->Right;
    vecZ = pStartNode->Direction;
    D3DXVec3Cross( &vecY, &vecZ, &vecX );

    // Generate the frame matrix for this branch
    D3DXMatrixIdentity( &mtxBranch );
    mtxBranch._11 = vecX.x; mtxBranch._12 = vecX.y; mtxBranch._13 = vecX.z;
    mtxBranch._21 = vecY.x; mtxBranch._22 = vecY.y; mtxBranch._23 = vecY.z;
    mtxBranch._31 = vecZ.x; mtxBranch._32 = vecZ.y; mtxBranch._33 = vecZ.z;
    mtxBranch._41 = pStartNode->Position.x;
    mtxBranch._42 = pStartNode->Position.y;
    mtxBranch._43 = pStartNode->Position.z;

    // Get the inverse matrix, to bring the node back into the frame's space
    D3DXMatrixInverse( &mtxInverse, NULL, &mtxBranch );
```

Now that we have a matrix that will transform vectors into the coordinate system of the branch, we will transform the direction vector, the right vector, and the position of the frond node into branch space by multiplying them with this matrix.

```
    // Build the axis in the frame's space
    D3DXVec3TransformNormal( &vecAxis, &pNode->Direction, &mtxInverse );
    D3DXVec3TransformNormal( &vecRight, &pNode->Right, &mtxInverse );
    D3DXVec3TransformCoord ( &vecPos, &pNode->Position, &mtxInverse );
```

At this point we only have the frond node's branch space position, look vector, and right vector. In order to position the vertices on the plane, we also need the second tangent vector (the plane's Up vector). We calculate this simply by rotating the branch space right vector around the direction vector (perpendicular to the plane) by 90 degrees.

```
    // Build the ortho vector which we use for our Y dimension axis
    D3DXMatrixRotationAxis( &mtxRot, &vecAxis, D3DXToRadian( 90.0f ) );
    D3DXVec3TransformNormal( &vecOrtho, &vecRight, &mtxRot );
```

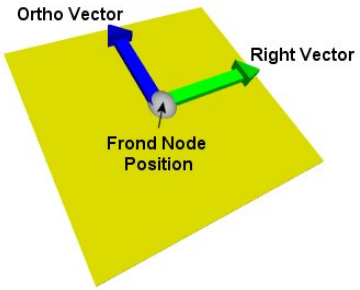

Figure 12.63

Figure 12.63 shows the plane and the two tangent vectors. Actually, because of the way we rotated our ortho vector it is actually pointing down and not up as shown in Figure 12.63, but hopefully you get the idea. As discussed in the AddBranchSegment function, we can combine the node position and the right and ortho vectors together with some scaling to position points anywhere on this plane. For example, we know that if we add the right vector to the node position, the resulting vector will be a point on the plane pointed to by the tip of the right vector in the diagram. If we add the ortho vector to the node position we would describe a position pointed to by the tip of the ortho vector in the diagram. If we add the negated right to the node position we would generate a point on the left side of the plane and similarly, if we add the negated ortho vector to the node position we would get a position at the bottom of the yellow plane shown in Figure 12.63. Those four position calculations we have just discussed describe the positions of four vertices on the

plane in the shape of a perfect cross. That is exactly where we want to base vertices of our intersecting quads to be. However, we do yet know how big we want these quads to be, so we will create some scaling values that will be multiplied with the ortho and right vectors so that we can create a larger or smaller cross on that plane.

In the next section of code we create three scaling values (fX, fY, and fZ) which will be used to scale the right, ortho and direction vectors before they are combined to create vertex positions. We will want the size of the fronds to get smaller as they are positioned higher up the tree and belong to smaller and thinner branches. We defined the minimum and maximum frond sizes as tree growth properties so we wish to find a size for the frond that is within this range but still related to the size of its parent branch. After all, a huge frond sticking out of a tiny parent branch would look unrealistic and quite bad. The technique we use is described next.

We will take the dimensions of the frond node (this was inherited from the parent when the frond node was created) and divide by the dimensions of the root node of the tree. As we know, the root node is the node in the tree with the largest dimensions, so this will create a value in the 0.0 to 1.0 range. The value is closer to zero the higher up the tree the frond is placed and the smaller its parent branch is in relation to the root node's dimensions. We will then use this scaling value to generate a new value between the minimum and maximum frond size growth properties that have been set. This is done on a per component level, so if we imagine that the variable ScaleValue describes the frond node's X dimension divided by the root node X dimension, fX would be calculated as:

fX = MinFrondSize + ((MaxFrondSize – MinFrondSize) * ScaleValue)

This is a basic interpolation calculation that we have used many times before. We are essentially adding the range (MaxFrondSize-MinFrondSize) of values that a frond can be set to, to the minimum frond size. This gives a value between MinFrondSize and MaxFrondSize based on ScaleValue (which is closer to zero the further from the root it is positioned). fX will then be used to scale the right vector of the frond node before it is added to the node position to generate the left and right vertex positions on the plane.

Below we see the code that calculates fX, fY and fZ.

```
    // If this is a frond node
    if ( pNode->Type == BRANCH_FROND )
    {
        // Calculate frond dimensions
        fScale = (pNode->Dimensions.x / m_pHeadNode->Dimensions.x);
        fX     = m_Properties.Frond_Min_Size.x +
                 ((m_Properties.Frond_Max_Size.x - m_Properties.Frond_Min_Size.x)
                  * fScale);

        fScale = (pNode->Dimensions.y / m_pHeadNode->Dimensions.y);
        fY     = m_Properties.Frond_Min_Size.y +
                   ((m_Properties.Frond_Max_Size.y - m_Properties.Frond_Min_Size.y)
                    * fScale);

        fScale = ((pNode->Dimensions.x / m_pHeadNode->Dimensions.x) +
                    (pNode->Dimensions.y / m_pHeadNode->Dimensions.y)) / 2.0f;
        fZ     = m_Properties.Frond_Min_Size.z +
```

```
                ((m_Properties.Frond_Max_Size.z - m_Properties.Frond_Min_Size.z)
                * fScale);
```

Notice that the scaling value used to create fZ is not calculated in the same way. That is, we do not scale the Z dimension of the frond node by the Z dimension of the root node. This is because, the fZ value will be used to control the length of the frond (how high the top of the quads are with respect to the node plane they are mounted on). We do not want this to be a product of the parent node's segment length but rather an average of the difference in the X and Y dimensions with respect to the root node. We would not want to have a really long frond branch if its X and Y dimensions are very small. This would create a skinny frond with a squashed texture.

Now that we have our Direction, Ortho, and Right vectors, our scaling values (fX,fY,fZ), and the node position, we can start building the vertex positions for those quads. We actually need to create four quads because we wish each of the polygons to be visible from both sides. Therefore, we will create four vertex positions per quad for a total of 16 vertex positions. Each quad will be formed from 4 vertices where two of the vertices lay on the plane and two are offset from the plane by some amount along the direction vector. The quads will be assigned their vertices such that they have a clockwise winding when viewed from the front.

In the next line of code we create a local array of 16 CVertex structures. In each element of the array we call the CVertex constructor that takes four parameters:

CVertex(D3DXVECTOR3 &vecPos, const D3DXVECTOR3 &vecNormal, float ftu, float ftv)

CVertex is defined in CObject.h and as you can see the first parameter is where we pass a 3D vector describing its position. The second parameter is where we pass in the normal of the vertex and the last two float parameters is where we pass in the texture coordinates. We will cover the first four vertex positions we add to this array one at a time so we can see how the quad is constructed. Here is the first section of the statement that defines the CVertex array and passes the first vertex.

```
// Calculate the actual frond vertices.
CVertex pVertices[16]={ CVertex( (vecPos - (vecRight*fX)), vecOrtho, 0.0f, 1.0f ),
```

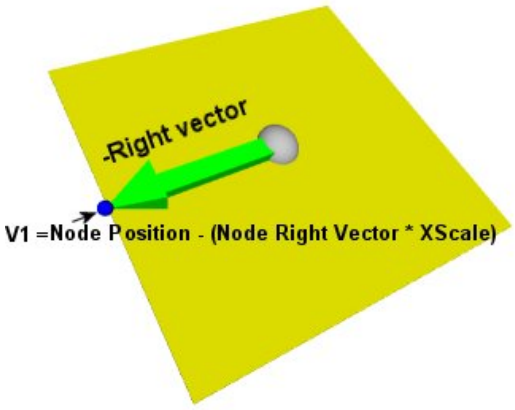

Figure 12.64

The position of the first vertex in the first quad is calculated by subtracting from the node position (shown as the ball in the center of the plane in Figure 12.64) the right vector scaled by the fX scaling value we calculated earlier. Essentially, fX dictates the width of the quad we are creating.

The resulting vertex position is situated on the plane and is shown as the small blue sphere labeled V1 in Figure 12.64. This is the bottom left corner of our first quad.

Notice that we pass in the vecOrtho vector as the vertex normal. Why? Because the right vector and the ortho vector are perpendicular to each other and therefore, in this diagram we can imagine that the ortho vector is actually pointing down towards the bottom edge of the yellow plane. This is the exact direction our quad

will be facing and thus its normal. Finally, since this is the bottom left vertex of our quad, we assign it the bottom left UV coordinate of the texture (0,1).

Let us see how we define the next (second) vertex in the array declaration.

```
CVertex( (vecPos - (vecRight * fX)) + (vecAxis * fZ), vecOrtho, 0.0f, 0.0f ),
```

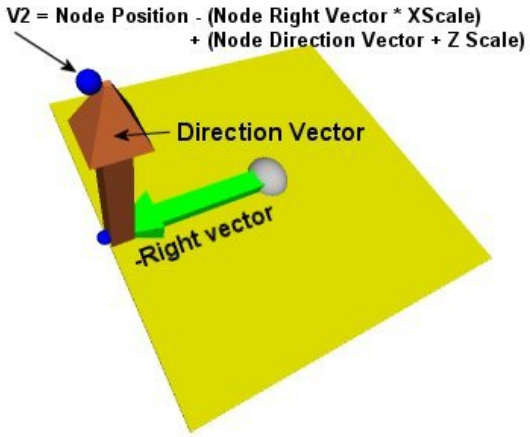

Figure 12.65

Here you can see that the next vertex position we define is initially calculated in the same way as before (by subtracting a scaled right vector from the node position) only this time we add to the resulting position the scaled direction vector (vecAxis). This vector is perpendicular to the plane and as such describes a position that is offset from the plane in the direction of the plane normal. This forms the top left corner of our quad. The same normal is used for every vertex in this quad.

As this is the top left vertex in the quad we assign it the texture coordinate for the top left of the texture (0,0).

It should be clear that in order to generate the top right and bottom right vertices we essentially do the same thing, only this time adding the right vector to the node position instead of subtracting it. This will form two symmetrical vertex positions on the right side of the node position.

```
CVertex( (vecPos + (vecRight * fX)) + (vecAxis * fZ), vecOrtho, 1.0f, 0.0f ),
CVertex( (vecPos + (vecRight * fX))                 , vecOrtho, 1.0f, 1.0f ),
```

Above we can see how the final two vertex positions are defined for the quad. These two calculations are identical to the previous two only with the addition of the right vector to the node position instead of its subtraction.

Note that it is the top right vertex we add third and the bottom right we add fourth, so that the four vertex positions of our quad describe a clockwise winding when viewed from the front.

Also notice that the two vertices are provided with the same vertex normal -- the ortho vector which is pointing toward us in this diagram. You should be able to see that the top right vertex's texture coordinate of (1,0) maps it to the top right of the texture image and the bottom left texture coordinate of (1,1) maps vertex V4 to the bottom right corner of the texture image.

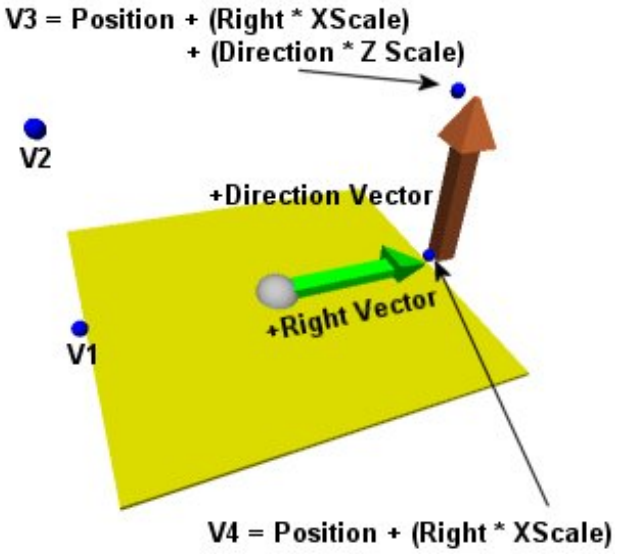

Figure 12.66

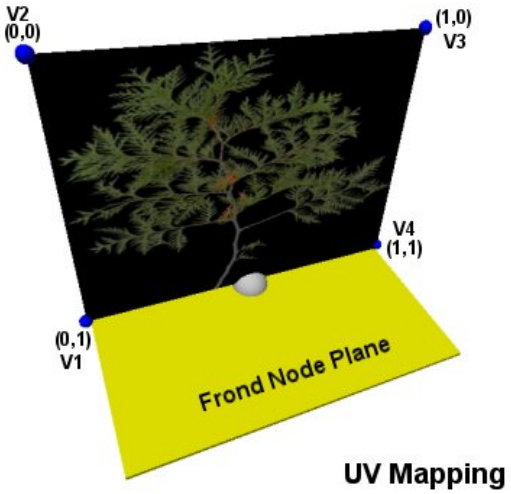

Figure 12.67

The next four vertices we define in the array describe the same quad, but facing in the other direction since we want this frond quad to be able to be viewed from both sides. We add the same four vertex positions, only this time in a counter-clockwise winding order. We also flip the sign of the vertex normals so that they are facing in the opposite direction.

```
CVertex( (vecPos + (vecRight * fX))                 , -vecOrtho, 1.0f, 1.0f ),
CVertex( (vecPos + (vecRight * fX)) + (vecAxis * fZ), -vecOrtho, 1.0f, 0.0f ),
CVertex( (vecPos - (vecRight * fX)) + (vecAxis * fZ), -vecOrtho, 0.0f, 0.0f ),
CVertex( (vecPos - (vecRight * fX))                 , -vecOrtho, 0.0f, 1.0f ),
```

We now have the horizontal quads built, so it is time to build the vertical ones so that we form a criss-cross shape of intersecting quads.

```
CVertex( (vecPos - (vecOrtho * fY))                 , -vecRight, 0.0f, 1.0f ),
CVertex( (vecPos - (vecOrtho * fY)) + (vecAxis * fZ), -vecRight, 0.0f, 0.0f ),
CVertex( (vecPos + (vecOrtho * fY)) + (vecAxis * fZ), -vecRight, 1.0f, 0.0f ),
CVertex( (vecPos + (vecOrtho * fY))                 , -vecRight, 1.0f, 1.0f ),
```

The vertex positions described above are shown in Figure 12.68. As you can see, this creates a quad aligned with the ortho vector instead of the right vector. As noted earlier, our ortho vector is actually pointing down and as such, the calculation of vertices V1 and V2 involve the subtraction of the ortho vector from the node position, while the calculations of vertices V3 and V4 involve its addition.

Notice that the vertex normal in each case we are passing is the negated right vector. The right vector is at right angles to the ortho vector and as such describes the plane of the quad we are building. The negated right vector will generate a vector pointing toward us in the diagram.

Finally, we want this quad to be two-sided so we add the same quad positions again, only this time with a reversed winding order and using the positive right vector as the vertex normals.

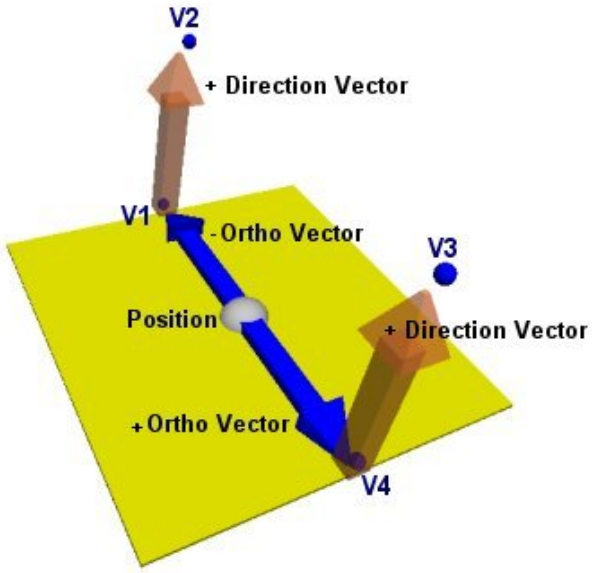

Figure 12.68

```
CVertex( (vecPos + (vecOrtho * fY))                  , vecRight, 1.0f, 1.0f ),
CVertex( (vecPos + (vecOrtho * fY)) + (vecAxis * fZ), vecRight, 1.0f, 0.0f ),
CVertex( (vecPos - (vecOrtho * fY)) + (vecAxis * fZ), vecRight, 0.0f, 0.0f ),
CVertex( (vecPos - (vecOrtho * fY))                  , vecRight, 0.0f, 1.0f )
};
```

We now have the 16 vertex positions stored in a CVertex array, so it is time to add them to the mesh. First we inform the passed CTriMesh of our intention to add 16 vertices so that it makes room at the end of its internal vertex array. We pass a pointer to the vertex array as well so that it can copy them.

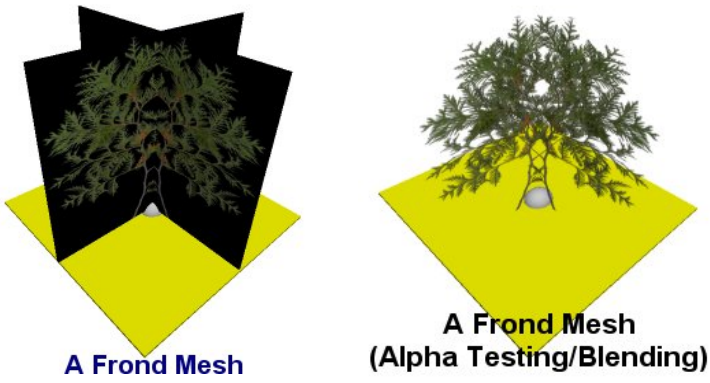

Figure 12.69

```
        // Add the frond vertex data
        long VIndex = pMesh->AddVertex( 16, pVertices );
        if ( VIndex < 0 ) return E_OUTOFMEMORY;
```

The AddVertex function returns the index (VIndex) of the first new vertex we added so that we know where we have to start indexing from when we build the indices.

For each quad we will need to add two triangles, each consisting of three indices (24 indices total). If we have a quad defined by vertices V1, V2, V3, and V4 in a clockwise winding order, then the first triangle will be created from V1, V2, and V3 and the second from vertices V1, V3, and V4. Below we see the code that allocates an array of 24 indices and indexes the vertices for each quad using this scheme. Of course, we must add VIndex to each zero based index value so that we index the positions in the mesh's vertex array where the frond vertices have been added. We cannot automatically assume they are the first 16 vertices in the mesh (since we know that they will not be because of prior branch vertex rings).

```
// Generate the index data for this frond
Index = (USHORT)VIndex;

USHORT pIndices[24] =
{ Index + 0 , Index + 1 , Index + 2 , Index + 0 , Index + 2 , Index + 3,
Index + 4 , Index + 5 , Index + 6 , Index + 4 , Index + 6 , Index + 7,
Index + 8 , Index + 9 , Index + 10, Index + 8 , Index + 10, Index + 11,
Index + 12, Index + 13, Index + 14, Index + 12, Index + 14, Index + 15 };
```

Finally, we inform the CTriMesh that we would like to add eight more triangles to its index buffer and pass the indices array so that it can copy them into its internal index array. Notice that in the third parameter to this function we pass in the number of the subset as 1 instead of 0. Thus, the branch segment faces will be assigned to subset 0 and the frond faces will be in subset 1. It makes perfect sense that we would need to put them in separate subsets since they both use different attributes (texture and material).

```
    // Add the frond face data (subset '1' )
    if ( pMesh->AddFace( 8, pIndices, 1 ) < 0 ) return E_OUTOFMEMORY;


    } // End if frond node

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

This function will be called to add a frond whenever a frond node is encountered during the mesh building process. As you have seen, the code is actually quite easy to follow.

The last function that needs a minor update is the BuildSkinInfo function which is called from BuildNode when a branch mesh has been fully populated with its vertex and index data and is about to be turned into a skin. This function creates the ID3DXSkinInfo object that will be passed to the CreateMeshContainer function and contains the connection between bones in the hierarchy and the vertices they influence.

CTreeActor::BuildSkinInfo - Updated

This function has been modified in two places to support fronds, but the modifications are very small. It is suggested that you follow this discussion with the source code to Lab Project 12.2 handy so that you can see the changes in context. We will not be showing the entire function here since this was a big function and very little has changed.

You will recall that this function has a very simple job. It traverses the branch (it is only ever called for BRANCH_BEGIN nodes) stepping from node to node within the same branch and adding the vertex indices it collects between bone nodes to the skin info object. The first change happens near the start of the function, where we execute a while loop to step through the current node's child list to find the next node that is a continuation of the branch. Originally, we just ignored any other BRANCH_BEGIN nodes since we were only interested in finding the next node in the current branch we were processing. However, now the child list of a node can contain both BRANCH_BEGIN nodes and BRANCH_FROND nodes, so an extra check has been added to ensure that we also ignore frond nodes.

```
    while ( pSearchNode = pSearchNode->Child )
    {
        // We're not interested in other branches or fronds, so shift us through
        // until we find a node that is our segment, or end node.
        while ( pSearchNode &&
               (pSearchNode->Type == BRANCH_BEGIN ||
                pSearchNode->Type == BRANCH_FROND) )
                pSearchNode = pSearchNode->Sibling;

        // If we couldn't find a node, we have an invalid hierarchy
        if ( !pSearchNode ) return D3DERR_INVALIDCALL;

        // Was a frame created here?
        if ( pSearchNode->BoneNode ) BoneCount++;

    } // Next child segment
```

The final alteration is at the very bottom of the function's inner loop. You will recall from our earlier discussion that this function, once finding a bone node, will start collecting the indices of all child nodes until the point that another bone is encountered. At this point all the indices currently collected in the temporary indices array are added for the previous bone in the ID3DXSkinInfo object. A new section has now been added which essentially just says, if we are processing a frond node, add the frond vertex indices to the temporary indices array also. If you have the source code in front of you, this addition will seem perfectly logical.

```
            else if ( pSearchNode->Type == BRANCH_FROND )
            {
                // Add influences for each of the 16 frond vertices
                for ( i = 0; i < 16; ++i )
                {
                    pIndices[ InfluenceCount ] = IndexCounter++;
                    pWeights[ InfluenceCount ] = 1.0f;
                    InfluenceCount++;
```

```
                } // Next index

            } // End if frond

        } // Next node sibling

        // If we couldn't find a segment node, we've reached the end of the branch
        if ( !pSegmentNode ) break;

    } // Next child segment

    // Clean up
    delete []pIndices;
    delete []pWeights;

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

We have now covered all the functions that need to be altered in the tree generation process and our CTreeActor class is now capable of creating trees with fronds. This certainly acts to increase our visuals by quite a significant margin.

However, we are not quite done yet in terms of our overall support for fronds. In the previous lab project, we could just allow the base class (CActor) DrawActor and DrawActorSubset methods to be called to render the tree, but this is no longer the case. We will need to override these functions so that when rendering frond subsets we can enable alpha blending and alpha testing before calling the base class versions of the function. Let us have a look at the CTreeActor::DrawActorSubset function first.

CTreeActor::DrawActorSubset

This function is extremely simple since it calls the base class implementation to do the actual rendering. All it does is test to see if the passed attribute ID matches the attribute ID of the frond subset (which we stored earlier in the m_nFrondAttribute member variable). If so, it enables alpha testing and alpha blending.

```
void CTreeActor::DrawActorSubset( ULONG AttributeID )
{
    // Is this our frond attribute?
    if ( m_Properties.Include_Fronds && AttributeID == m_nFrondAttribute )
    {
        // Setup states
        m_pD3DDevice->SetRenderState( D3DRS_ALPHATESTENABLE, TRUE );
        m_pD3DDevice->SetRenderState( D3DRS_ALPHAREF, (DWORD)0x000000A0 );
        m_pD3DDevice->SetRenderState( D3DRS_ALPHAFUNC, D3DCMP_GREATEREQUAL );
        m_pD3DDevice->SetRenderState( D3DRS_ALPHABLENDENABLE, TRUE );
        m_pD3DDevice->SetRenderState( D3DRS_SRCBLEND, D3DBLEND_SRCALPHA );
        m_pD3DDevice->SetRenderState( D3DRS_DESTBLEND, D3DBLEND_INVSRCALPHA );
        m_pD3DDevice->SetTextureStageState( 0,D3DTSS_ALPHAOP, D3DTOP_SELECTARG1 );
```

```
        m_pD3DDevice->SetTextureStageState( 0, D3DTSS_ALPHAARG1, D3DTA_TEXTURE );

    } // End if frond attribute

    // Call underlying draw
    CActor::DrawActorSubset( AttributeID );

    // Reset states if applicable
    if ( m_Properties.Include_Fronds && AttributeID == m_nFrondAttribute )
    {
        m_pD3DDevice->SetRenderState( D3DRS_ALPHATESTENABLE, FALSE );
        m_pD3DDevice->SetRenderState( D3DRS_ALPHABLENDENABLE, FALSE );

    } // End if frond attribute
}
```

We first enable alpha testing because alpha blending by itself will not suffice. We do not want the black background pixels to be rendered at all. Although these pixels have zero alpha components and would not alter the contents of the frame buffer when rendered, with alpha blending enabled, they would actually still be rendered. Although these pixels would be blended with the frame buffer with zero weight (leaving the frame buffer unchanged) their depth values would still be written to the depth buffer. Therefore, the black space around the leaves would block leaves located behind from being drawn as they would be rejected by the depth test. In essence, it would seem as if the leaves were blocked by invisible geometry.

Figure 12.70 : Low Alpha Testing Reference Value (Ref = 5)

By enabling alpha testing we can make sure that these background pixels get rejected in the pixel pipeline. This means their depth values will not be recorded in the depth buffer and therefore, we will not have these issues. We set the alpha testing reference value to a rather arbitrary 0xA0 (160) so that all pixels with an alpha value of less than 160 get rejected. The reason we do not set this to a value such as 1 for example (so only totally transparent pixels such as the background pixels get rejected) is because, when we generate an alpha channel, the pixels between the opaque leaf edges and the transparent black background will often also be blended from opaque to transparent. Therefore, this makes sure that we remove nearly all the pixels that are mostly transparent, but leave the ones in place that are partially transparent since these pixels may contain part of the leaf edge. Setting the alpha reference to a low value (e.g., 1) will remove all the background pixels but will leave the partially transparent pixels around the opaque leaves in place and their depth values will still be written to the frame buffer. This provides an unattractive border around the leaves (see Figure 12.70). As you can see, there is a large number of pixels that are not totally transparent that do not get rejected by the alpha testing pipeline, so setting a higher reference value helps filter these undesirables out.

Figure 12.71 Alpha Blending Disabled

We then enable alpha blending so that the leaves get blended with the frame buffer and any partially transparent pixels around the leaf edges which did not get rejected from the alpha test will allow for objects behind them to show through. If you play around with the alpha testing reference value you will see exactly why this is necessary. If we do not enable alpha blending, the results will not look overly bad, but there will still be a hard edge to leaves (see Figure 12.71). Enabling alpha blending allows us to keep the outer portions of the leaves that are partially transparent but have them blend out smoothly, removing the hard edge around the leaves. You can experiment with the settings to find what works best for you. Obviously, while alpha blending does seem to produce better results, it adds some additional cost.

CTreeActor::DrawActor

The CActor::DrawActor function can be used to automate the rendering of managed mode meshes. It is called once by the application and will render all of its subsets. It can do this because a managed mode mesh contains its own texture and material arrays and can set them before rendering each of its subsets.

We also have to override this function in CTreeActor for the same reason as before (to enable the alpha testing and alpha blending render states). Unfortunately, when rendering a managed mode tree we no longer have per-subset control to decide whether we should enable the states for the entire rendering of the tree or not. This means that if the tree does have fronds, we will have to enable alpha testing and alpha blending and then use the base class implementation to instruct the mesh to draw all its subsets. Sadly, this does mean that both subsets will be rendered with these states enabled. These states will not harm the rendering of the non-frond subset, but it is not ideal to have these states set when we do not wish to use them since they introduce overhead in the pixel pipeline.

```
void CTreeActor::DrawActor( )
{
    // Fronds included?
    if ( m_Properties.Include_Fronds )
    {
        // Setup states
        m_pD3DDevice->SetRenderState( D3DRS_ALPHATESTENABLE, TRUE );
        m_pD3DDevice->SetRenderState( D3DRS_ALPHAREF, (DWORD)0x000000A0 );
        m_pD3DDevice->SetRenderState( D3DRS_ALPHAFUNC, D3DCMP_GREATEREQUAL );
        m_pD3DDevice->SetRenderState( D3DRS_ALPHABLENDENABLE, TRUE );
        m_pD3DDevice->SetRenderState( D3DRS_SRCBLEND, D3DBLEND_SRCALPHA );
        m_pD3DDevice->SetRenderState( D3DRS_DESTBLEND, D3DBLEND_INVSRCALPHA );
        m_pD3DDevice->SetTextureStageState( 0,D3DTSS_ALPHAOP, D3DTOP_SELECTARG1 );
        m_pD3DDevice->SetTextureStageState( 0, D3DTSS_ALPHAARG1, D3DTA_TEXTURE );

    } // End if fronds included

    // Call underlying draw
    CActor::DrawActor();
```

```
    // Reset states
    if ( m_Properties.Include_Fronds )
    {
        m_pD3DDevice->SetRenderState( D3DRS_ALPHATESTENABLE, FALSE );
        m_pD3DDevice->SetRenderState( D3DRS_ALPHABLENDENABLE, FALSE );

    } // End if fronds included
}
```

Conclusion

This concludes our discussion of trees (for now) and also brings us to the end of our skinning discussions in this course. There was quite a lot of challenging material to take in at times, but you should now have a very strong understanding of frame hierarchies and skinned meshes. We now have an actor that fully supports skinning which we can use in all of our applications from this point on. Additionally we have a pretty nice derived actor class that allows us to generate trees for our various scenes. We recognize that there was a lot to digest in this chapter, so take your time and re-read the sections that you found difficult before moving on to the workbook.