Virtual TreeView tutorial

Virtual TreeView Tutorial

The versitale Delphi component

---

by Philipp Frenzel

Table of Contents

Installation

Part I

1 Installation

Autor: Philipp Frenzel
Translated by: Koos de Graaf

disclaimer by Koos:
I do not know Philip Frenzel or Mike Lischke and I did not asked them permission to translate this tutorial.
The original text is all by Philipp Frenzel, I tried to stay as close to his word but sometimes I have elaborated a bit or changed the structure of the sentence.
I did the best I could translating it but sometimes I'm not sure. In those cases I left my best guess and the original text for you.
I hope you will enjoy using, I know many people where waiting for a translation.
Good luck,

Koos

In your daily computer business you will often run into components that display their information in a tree-top or directory structure. The Windows explorer for example displays hard disks, files and folders in this manner. In Delphi there is a component that controls this Windows-Control: TTreeview. For smaller tree-top structures this component is adequate and up to the task, but somewhere along the road there will come a point when you need more: columns, better performance, higher flexibility, Windows XP styles, Unicode support, etc.

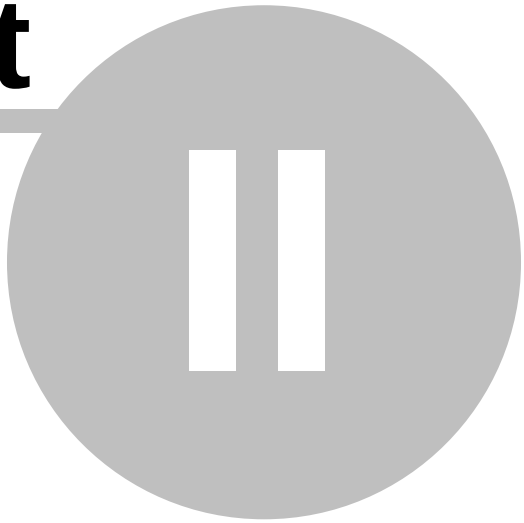
It is time to find yourself a new component that has those functionalities, like for instance the VirtualTreeView component of Mike Lischke. This component is an Open Source project. You can download the official release from Mike's Homepage. You will also find more information about this incredible component. For help and support Mike recommends its homepage, the VT Newsgroup. Many possibilities of the tree are being shown in the demo's, which you can also find on his homepage. Visit the picture gallery to get an impression of al the possibilities of the VirtualTrees. The components do not need runtimes, special DLL's or other external tools. The source code is directly linked in the EXE data.

After the download, which is about a 2.2MB archive (except for the Controls, the complex HTML-help file en the demo source code), the component must be installed. You can do this by using the packages or the VirtualTreesReg,pas, which also contains the register procedure. Make sure the search path will not contain an old version of the components! After the installation you will find tree components in the "Virtual Controls" tab: VirtualStringTree (VST), VirtualDrawTree and the newest HeaderPopmenu. The important component for us is the VST.

Notice: The component can be used from Delphi version 4.

It will be impossible to describe all functions and possibilities in the VirtualTrees. It is therefore helpful to have at least same experience with the VirtualTree component.

Part



2 The first node

The units of a Tree are called Nodes. The first level of a Tree is the root level. Sub nodes are called Children or Child Nodes.

The creation of a new node is very simple:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  VST.AddChild(nil);
end;
```

The Method will transfer the parameter nil, because this node is a root node. This is where you will assign the parent node. The node will then be attached as a child of that node.

Now we will create a new node, which has a new node as a child.

```
procedure TForm1.Button1Click(Sender: TObject);
var
  Node: PVirtualNode;
begin
  Node:=VST.AddChild(nil);
  VST.AddChild(Node);
end;
```

The complete tree should look like this (after one call to this function).

Explanations:

```
Node := VST.AddChild(nil);
```

The Node is now shown from the root node. You will need the memory address, so you can add 'Children' later. We can provide this address by using the Node variable as parameter in the second call to the function.

PVirtualNode is a pointer to the record TVirtualNode, which will hold some information of the focussed Node.

Another possibility, for instance to immediately create 100 root nodes, is to set the property RootNodeCount of the Trees to 100. From that moment on all the Nodes will have the caption 'Node'. This is normal. How you can change this, I will explain later.

2.1 React on nodes by clicks

Now one click should activate an appropriate action. First we will add a couple of Nodes:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  Node: PVirtualNode;
  I: Integer;
begin
  Node:=VST.AddChild(nil);
  for I:=1 to 100 do
    Node:=VST.AddChild(Node);
end;
```

In this example 101 nodes are being created on 100 levels. When a node is clicked the program should display on which level the node has been created. So we will write the following in the OnClick event of the tree:

```
procedure TForm1.VSTClick(Sender: TObject);
var
  Node: PVirtualNode;
begin
  Node:=VST.FocusedNode;
  if not Assigned(Node) then
    Exit;

  Showmessage(IntToStr(VST.GetNodeLevel(Node)));
end;
```

The property of FocusedNode contains a pointer the node selected at that time. If no node is selected, the property will be nil.
The method GetNodeLevel of the tree will return the level the node is on. As parameter you must enter a pointer to the node of which the level will be returned.

You have just learned that you can created a high number of root nodes by setting the property RootNodeCount. With the help of the property ChildCount of the tree, you can also set the number of children:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  Node: PVirtualNode;
  I: Integer;
begin
  for I:=1 to 100 do
  begin
    Node:=VST.AddChild(nil);
    VST.ChildCount[Node]:=10;
  end;
end;
```

In this case every one of the 100 root nodes will 'receive' 10 children. When using this way to create the root nodes and the child nodes you must realize you will not always have the opportunity the assign an object to a child node. More about that later...

2.2 Insert nodes in a specific location

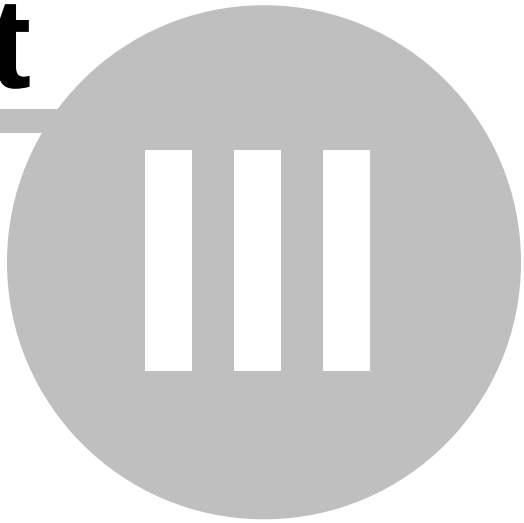
With the help of the method InsertNode it is possible to insert a node in a specific location. The method will need to parameters. First the address of the focussed node in the form of PVirtualNode and the insert mode:

```
Node := vst.InsertNode(vst.FocusedNode, amInsertAfter);
```

In this example the node is inserted after the focussed node. Would the parameter be 'amInsertBefore', the node would be inserted before the focussed node.

Insert node data

Part



3 Insert node data

To this moment inserting a node was relatively easy. From this moment on it will be a little more complex. Not only will we add a note but also all the data we want to attach to it.

3.1 Define the Data type

You can add your data quite nicely in the form of a record. For that purpose we will define a record, which could look a bit like this:

```
type
  PTreeData = ^TTreeData;
  TTreeData = record
    FCaption: String;
    FColumn1: String;
  end;
```

The pointer PTreeData will point to the record. Why we will use a record in this way, I will explain later. You can add your own variables to the record.

3.2 Inserting the data

The next lines are meant to provide an insight in the TreeView component and don't necessarily are part of the source code.

A node is inserted as usual. The address of the node is stored in a variable locally:

```
var
  Node: PVirtualNode;
begin
  Node:=VST.AddChild(nil);
end;
```

By now this should look familiar. In the next piece of code you will retrieve the position of the object (the last object will until then be nil). We achieve that like this:

```
Data := VST.GetNodeData(Node);
```

Data must by of the type PTreeData. PTreeData points once again to the record TTreeData (so Data:PTreeData;). Now we will fill the record with values:

```
Data^.FCaption := 'Hallo';
Data^.FColumn1 := 'Weiterer Wert';
```

The record has now been filled with data. This data is now connected to the node. But there are still one or two traps we will have to avoid. The size of the structure must be known to the tree. You can assign it by using:

```
VST.NodeDataSize := SizeOf(TTreeData);
```

You will normally only assign this once. Only if you radically change the build-up of your data (and thereby the size) you will have to assign the NodeDataSize again. Alternatively you can use the event OnGetNodeDataSize, which will be fired every time the data size is been asked for.

The complete procedure of inserting the data in a node looks like this. I put the actual inserting in a function:

```
function AddVSTStructure(AVST: TCustomVirtualStringTree; ANode:
PVirtualNode;
  ARecord: TTreeData): PVirtualNode;
var
  Data: PTreeData;
begin
  Result:=AVST.AddChild(ANode);
  Data:=AVST.GetNodeData(Result);
  Avst.ValidateNode(Result, False);
  Data^.FCaption:=ARecord.FCaption;
end;

procedure TForm1.Button3Click(Sender: TObject);
var
  I: Integer;
  TreeData: TTreeData;
begin
  VST.NodeDataSize:=SizeOf(TTreeData);
  VST.BeginUpdate;
  for I:=0 to 100 do
  begin
    TreeData.FCaption:='Node-Nummer: '+IntToStr(I);
    AddVSTStructure(VST,nil,TreeData);
  end;
  VST.EndUpdate;
end;
```

In this example 100 nodes and their data are inserted.

The function AddVSTStructure has three parameters. The first is the tree, in which the node will be inserted. Then secondly the parent-node (if a root node must be added, this parameter should be set to nil) and the third parameter is the record. As result the function will give back - precisely as AddChild - a pointer to the inserted node.

The method ValidateNode will initialise the nodes. This is important at the freeing of the data, because otherwise the data will be released into the memory. The first parameter must be the node you want to initialise, the second if you also want to initialise its children. We will not need this in our case.

3.3 Read out the data

OK. We now have inserted the data, but don't know how to read out the information in it. Here is an example.

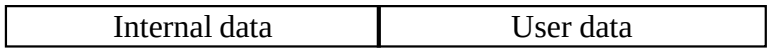
```
procedure TForm1.VSTClick(Sender: TObject);
var
  Node: PVirtualNode;
  Data: PTreeData;
begin
  Node:=VST.FocusedNode;
  if not Assigned(Node) then
    Exit;

  Data:=VST.GetNodeData(Node);
  Showmessage(Data.FCaption);
end;
```

One part you will recognise from the last chapter. It will set the focused node in the OnClick-Event of the tree.
New are the two last parts. The variable Data is again a pointer to the record TTreeData in the form of PTreedata. With the function GetNodeData this variable is set to the position of the record of the node that you use as parameter (in this case Node). Using Showmessage we will show the value FCaption.

3.4 A little bit of theory

When you insert a node the memory will look a bit like this:



First the internal node information is read, then the user data. Because the user data is dynamic, you must specify the size beforehand. As said we can do this with the property NodeDataSize. The function GetNodeData will calculate the approximated start-value of the user data.

3.5 Releasing the data

Earlier I have discussed releasing the data. The data occupies a reserved space in the memory that must be freed after exiting. This does not happen automatically, so it must be done manually. The event OnFreeNode is the right way to do it:

```
procedure
TForm1.vstFreeNode(Sender:
TBaseVirtualTree; Node:
PVirtualNode);
var
  Data: PTreeData;
begin
  Data:=VST.GetNodeData(Node);
  if Assigned(Data) then
    Data.FCaption:='';
end;
```

By using GetNodeData we are getting the memory address of the data. With strings it is enough to set the variable to an empty string.

Part IV

4 The node caption

With the normal Treeview component it was simple: There was a direct routine (AddText), wich you could use immediately at the moment you inserted a node. It is not so simple with VirtualTree. But in return the inserting process is much faster.

This chapter is larger leaning on the knowledge you would have gained from the last chapter. A record is once again added to the node. The variable FCaption in the record will be used for the caption.

Now we must place before and after the FOR-statement a VST.BeginUpdate and (after) a VST.EndUpdate. With this you will signify that the tree is being edited and Windows will suppress the repainting of the tree.

4.1 The OnGetText event

This event will be called at the moment a node is painted. Therefore this is the place to set the caption (with the reference parameter CellText).

In our example this would look like this:

```
procedure TForm1.VSTGetText(Sender: TBaseVirtualTree; Node:
PVirtualNode;
  Column: Integer; TextType: TVSTTextType; var CellText: WideString);
var
  Data: PTreeData;
begin
  Data:=VST.GetNodeData(Node);

  CellText := Data^.FCaption;
end;
```

Note: In the older versions the parameter CellText was called Text (also still in some places in the manual, I think...)

The caption of every node will then have the value, which is in the variable FCaption of the record (it should be a bit like: Node-Nummer: xxx). If the node can not retrieve data, the program will crash.

You should not make big or complex calculations in this event. Just the retrieving of user data.

This is where you can download the example program of the last chapter.

Working with Coloumns

Part V

5 Working with Coloumns

Another more clarifying difference with the normal TreeView component is the possibility to create a tree-top structure with the help of columns.
Choose the property 'Header' in the object-inspector. Then set the different columns by using the property-editor. In the options menu of the header property you must activate 'hoVisible'.

Of course you can set the caption property of a header with the object-inspector, but there will come a time you will need to set at run-time:

```
procedure TForm1.Button2Click(Sender: TObject);
begin
  VST.Header.Columns[0].Text:='Erste Spalte';
end;
```

This will set the caption of the first column to 'Erste Spalte' (First column in German). With the property width you can set the width of the column. To make a column invisible use the following code:

```
procedure TForm1.Button2Click(Sender: TObject);
begin
  VST.Header.Columns[0].Options:=VST.Header.Columns[0].Options -
[coVisible];
end;
```

The column will be visible again when you change the '-' into a '+'.

By default the column act like buttons. Test it yourself if you have compiled the program and click with your mouse on the column. With that action, two events can be called: OnColumnClick or OnColumnDblClick. The will be called with a simple click, the other with a double-click. The procedure header can look like this:

```
procedure TForm1.vstColumnClick(Sender: TBaseVirtualTree; Column:
Integer;
  Shift: TShiftState);
```

The parameter Column has the Column index, of the column that was clicked.

If you want the headers to have a static caption, like with StringGrid you must remove the coAllowclick option out of the header.

5.1 Nodes and Columns

Now you can attach more column values to a node. We have granted the record TTreedata from chapter 3 another variable.

```
type
 PTreeData = ^TTreeData;
 TTreeData = record
  FCaption: String;
  FColumn1: String;
 end;
```

In the procedure, that is added to the tree of the node, we must fill these variables with values that are also are attached to the data-record in the function AddVSTStructure.

```
function AddVSTStructure(AVST: TCustomVirtualStringTree; ANode:
PVirtualNode;
  ARecord: TTreeData): PVirtualNode;
var
  Data: PTreeData;
begin
  Result:=AVST.AddChild(ANode);
  Data:=AVST.GetNodeData(Result);
  AVST.ValidateNode(Result,False);
  Data^.FCaption:=ARecord.FCaption;
  Data^.FColumn1:=ARecord.FColumn1;
end;

procedure TForm1.Button3Click(Sender: TObject);
var
  I: Integer;
  TreeData: TTreeData;
begin
  VST.NodeDataSize:=SizeOf(TTreeData);
  VST.BeginUpdate;
  for I:=0 to 100 do
  begin
    TreeData.FCaption:='Node-Nummer: '+IntToStr(I);
    TreeData.FColumn1:='Nummer: '+IntToStr(I*I);
    AddVSTStructure(VST,nil,TreeData);
  end;
  VST.EndUpdate;
end;
```

Note: This procedure can be taken directly from the old example.

We still have to modify the OnGetText-event. If you look at the parameter list of the event procedure, you will also see the parameter 'Column' which is the column index.

```
procedure TForm1.VSTGetText(Sender: TBaseVirtualTree; Node:
PVirtualNode;
  Column: Integer; TextType: TVSTTextType; var Text: WideString);
var
  Data: PTreeData;
begin
  Data:=VST.GetNodeData(Node);

  case Column of
    0: Text := Data.FCaption;
    1: Text := Data.FColumn1;
  end;
end;
```

The case statement is needed. Just like with many elements the count starts at 0. The first column is also 0. If no column is defined (which is the default) the parameter will be -1. There has to be at least one column defined or we will get in trouble with selecting the node in time.

Surely we must not forget to free the data. Use the procedure in chapter 3 12.

5.2 General Header events

On the events tab of the object-inspector are multiple events that are connected to the header. They all start with OnHeader... . A short note on those events:

OnHeaderClick & OnHeaderDblClick
: These events are called when the user (double)clicks a certain header.

OnHeaderDragged & OnHeaderDragging
: The VirtualTrees have multiple methods and elements to support drag & drop. You can for instance change the way the header (splitters, columns, rows or headers?) are ordered. OnHeaderDragging is called before a Drag event is completed (which can then be suppressed by the parameter Allowed) and OnHeaderDragging is called after the Drop action. If you want to disable drag & drop from the get-go, use the 'hoDrag' value of the header property of the tree.

OnHeaderDraw
: If the header is painted, this event will be called.

OnHeaderMouseDown, OnHeaderMouseMove & OnHeaderMouseUp
: These events share a strong resemblance with OnMouseDown, OnMouseMove and OnMouseUp, who are all descendants of TComponent.
OnMouseHeaderDown is called when the user clicks on an area of the header, OnHeaderMouseMove when the mouse moves in an area of the header and OnHeaderMouseUp, when the mouse button is released (in the header area?).

Adding a object to the node

6 Adding a object to the node

To add a object to a node we must act like we would if we are adding data to a node. One variable of the type TObject in the record will simply point to our object. It's that simple.

6.1 Define an object

First we must define and declare our object. As object we will use a class.

```
type
  TTreeDataClass = class
    private
      FTestStr1: String;
      FTestint: Integer;
    published
      property TestStr1: String read FTestStr1 write FTestStr1;
      property TestInt: Integer read FTestInt write FTestInt;
  end;
```

If you want you can use many types as properties. It is the same as defining a component. Because it is just a normal class, you will have all the possibilities of a normal class at your disposal. Not only can you declare properties, but you can also integrate functions and procedures in the class. This class has two properties 'TestStr1' and an integer property 'TestInt'.

For this type we will declare the next record:

```
type
  PTreeData = ^TTreeData;
  TTreeData = record
    FObject : TObject;
  end;
```

After this the pointer PTreeData will point to the record. Why we are using a record here, I will explain in the next chapter. By now you must suspect that the variable 'FObject' of the record will later contain the data of our declared object!

6.2 Insert the object

The following lines are mainly used to provided same insight and are not always part of the source code.

Add a node like we would normally do. The address of the node will be held in a local variable. You should knows this by now, I told you in chapter 3.

```
var
  Node: PVirtualNode;
begin
  Node:=VST.AddChild(nil);
  Data:=VST.GetNodeData(Node);
end;
```

Data must in this case be a PTreeData type. PTreeData will point at the class TTreeData (so Data: PTreeData;). Now we will assign a new object to the variable FObject:

```
Data^.FObject:=TreeDataClass;
```

TreeDataClass is in this case in instant of the class TTreeDataClass (so TreeDataClass: TTreeDataClass).
The Object should now be assigned.

The complete procedure to add a node with an object looks like this. The actual adding has been put into a function:

```
function AddVSTObject(AVST: TCustomVirtualStringTree; ANode:
PVirtualNode;
  AObject: TObject): PVirtualNode;
var
  Data: PTreeData;
begin
  Result:=AVST.AddChild(ANode);
  AVST.ValidateNode(Result,False);
  Data:=AVST.GetNodeData(Result);
  Data^.FObject:=AObject;
end;

procedure TForm1.Button1Click(Sender: TObject);
var
  I: Integer;
  TreeObject: TTreeDataClass;
begin
  VST.NodeDataSize:=SizeOf(TTreeData);
  for I:=1 to 100 do
  begin
    TreeObject:=TTreeDataClass.Create;
    try
      TreeObject.TestStr1:='Node-Nummer: '+IntToStr(I);;
      TreeObject.TestInt:=Random(1000);
      AddVSTObject(VST,nil,TreeObject);
    except
      TreeObject.Free;
    end;
  end;
end;
```

In this example 100 nodes + objects are added. The object has to be created every time. You may wonder why the object is freed in a except-block. There is a very simple reason why. The object will exist the entire time and not be destroyed after the procedure has finished. We will worry about freeing the object later. The object has to be freed manually when an error has occurred during the adding of a node (for instance, out of memory).

The function AddVSTObject awaits 3 parameter. Firstly the tree, to which a node will be added. Then the as second parameter, the parent node (if a root node has to be added, this parameter must be nil) and as third parameter the object. As Result the function - like AddChild - will point to the added node.

6.3 Read the object

Great. Now that we have added an object, but we don't know how we can read it out. Here is an example.

```
procedure TForm1.VSTClick(Sender: TObject);
var
  Node: PVirtualNode;
  Data: PTreeData;
begin
  Node:=VST.FocusedNode;
  if not Assigned(Node) then
    Exit;

  Data:=VST.GetNodeData(Node);
  Showmessage(TTreeDataClass(Data.FObject).TestStr1);
end;
```

One part of the procedure you will recognise from the last chapter. The focused node is searched for in the tree.

The two last parts are new. The variable Data is once again a pointer to the TTreeData class in the form (or disguise) of PTreeData. With the function Get NodeData the position of the object of the selected node is retrieved. By typecasting the class is accessed and the contains of property TestStr is shown in Showmessage.

So adding an object barely differs from adding normal data.

6.4 Releasing

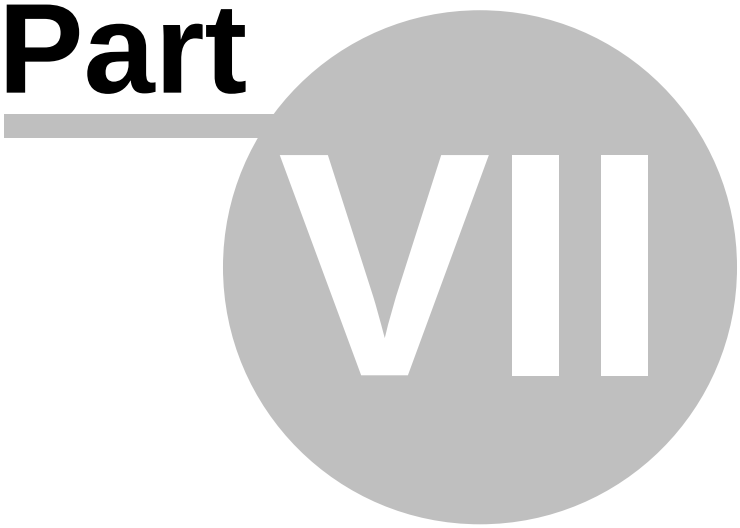
Like the data we must also release the object and set it's pointers to nil. The tree has prepared the event OnFreeNode for these cases, which will be called when the node is released. Although the node is released, the data will not be released automatically. We will need to do it by code.

```
procedure TForm1.vstFreeNode(Sender: TBaseVirtualTree; Node:
PVirtualNode);
var
  Data: PTreeData;
begin
  Data:=VST.GetNodeData(Node);
  if not Assigned(Data) then
    exit;

  Data.FObject.Free;
end;
```

An example of this chapter can be downloaded here.

Other Commands



7 Other Commands

In one tutorial you can't discuss every function of VirtualTreeView. In this chapter I will however talk about the methods and functions that are the most important and most used. People transferring from the TTreeview component will soon find out that the commands that are available in the TTreeNode object are also available in the VirtualTrees. It is just that most of the methods are not available in the node but in the tree.

7.1 Foccused node

This command is already used a few times in this tutorial.

```
procedure TForm1.VSTClick(Sender: TObject);
var
  Node: PVirtualNode;
begin
  Node:=VST.FocusedNode;
  if not Assigned(Node) then
    Exit;
end;
```

The important method is the FocusedNode method. Because it is usually in the OnClick-event of the tree, it is of course possible that the user will click on an area with no node. To prohibit a crash we need the If-statement. The procedure is ended when there no node selected.

7.2 Retrieve the node level

The function GetNodeLevel will return the level of a node. The highest level is 0. As parameter a node of the PVirtualNode type must be given.

```
VST.GetNodeLevel(Node);
```

7.3 Delete a node

Until this moment we have only added nodes. But somewhere along the line you will have to delete a node. You can do this by using the method DeleteNode, which will need the node of the PVirtual type.

```
VST.DeleteNode(Node);
```

7.4 Delete all the children of a node

In our example we will use this tree:

```
Node1
  |
  |__Node2
  |  |
  |  |__Node3
  |  |  |
  |  |  |__Node4
  |  |
  |  |__Node5
  |
  |__Node6

Node7
  |
  |__Node8
  |  |
  |  |__Node9
  |
  |__Node10
```

Now we would like to delete all the children of node 2. The effected need would be node 3, node 4 and node 5. This is how you will delete the nodes:

```
VST.DeleteChildren(Node);
```

Again the node must be a PVirtual type and point to node 2.

7.5 The position of a node

The method AbsoluteIndex will return the Index of a node. The nodes are globally sequentially numbered from top to bottom.
You can see the function of AbsoluteIndex really well when you try this next example:

```
procedure TForm1.VSTGetText(Sender: TBaseVirtualTree; Node:
PVirtualNode;
  Column: Integer; TextType: TVSTTextType; var Text: WideString);
begin
  Text:=IntToStr(VST.AbsoluteIndex(Node));
end;
```

Just insert a tree structure by using AddChild and write the other lines in the OnGetText-event. The method AbsoluteIndex will require a node of the PVirtualNode type.

7.6 Expand / Collapse all nodes

TVirtualTreeView has provided two methods which will execute these actions. The try this, first insert a couple of nodes.

```
VST.FullExpand;
```

The method FullExpand will expand all the nodes. On the actual expanding the OnExpanding-event is called for each node, in which you can, via the parameter Allowed, decide is the will be expanded. After that (if you let the node expand) the event OnExpanded will follow.

```
VST.FullCollapse;
```

The method FullCollapse will collapse all the nodes. Also in this case two events are called for each node, OnCollapsing (before) and OnCollapsed (after).

7.7 Expand / Collapse a specified node

With the help of the property Expanded of the tree, you can expand a specified node.

```
VST.Expanded[Node]:=True;
```

Between the square brackets is a variable of the PVirtualNode type. Is the property set to true, the node is expanded, if it is set to false it is collapsed. Also in this case the two events On...ing and On...ed will be called, where '...' will be the applicable action (for example On**Expand**ing).

7.8 Does a node have children?

The tree property HasChildren will tell you if a node has children or not.

```
if VST.HasChildren[Node] then
  Close;
```

In the square brackets is once again our variable of the PVirtualNode type. Is the property set to true, the node has children, otherwise it will be set to false.

7.9 Delete all Nodes

You can do this using the method Clear:

```
VST.Clear;
```

To speed up the process you should deactivate the repainting of the control with VST.BeginUpdate and reactivate it later with VST.EndUpdate.

7.10 Determine the parent

Every node (PVirtualNode) has the property parent. This property points to the parent of the node. The parent property of a root-node will point to the tree. To differentiate a root-node from a normal node you can use the following:

```
var
  Node: PVirtualNode;
begin
  if not Assigned(vst.FocusedNode) then
    exit;

  Node:=vst.FocusedNode;
  while vst.GetNodeLevel(Node)>0 do
    Node:=Node.Parent;
end;
```

The variable node will point exclusively to the root-node of a selected node.

7.11 Nodes with different heights

By using the property NodeHeight you will set the height of a node. The selected node must be used as Index. In this example the height is set in the OnInitNode event.

```
procedure TMainForm.vstInitNode(Sender: TBaseVirtualTree;
  ParentNode, Node: PVirtualNode;
  var InitialStates: TVirtualNodeInitStates);
begin
  vst.NodeHeight[Node]:=Random(10)+15;
end;
```

You might have noticed that the record node has a variable NodeHeight. Nevertheless don't use this variable to set the height, but use the code in this example.

7.12 Object Hierarchie

The classes of the tree have the following hierarchy:

- TCustomControl
  - TBaseVirtualTree
    - TCustomVirtualDrawTree
      - ❖TVirtualDrawTree
    - TCustomVirtualStringTree
      - ❖TVirtualStringTree

By rule the Sender parameter in the event-handler of the TBaseVirtualTree type. The same handler can then be used for the normal TVirtualStringTree and for the TVirtualDrawTree.

Sorting nodes

8 Sorting nodes

The tree makes the sorting of nodes very simple. By using the method SortTree the tree is sorted. A call to this method could look like this:

```
vst.SortTree(0, sdAscending, True);
```

The first parameter is the column, that will be sorted. The second parameter determines the sort orientation, sdAscending will set it to sort ascending (A -> Z), sdDescending will set it to sort descending (Z -> A). The third parameter is optional and is set to true as default.

To ensure the sorting procedure will finish successfully, a comparing procedure must be implemented. That is why the tree will have the OnCompareNodes event:

```
procedure TForm1.vstCompareNodes(Sender: TBaseVirtualTree; Node1,
  Node2: PVirtualNode; Column: Integer; var Result: Integer);
```

Node1 and Node2 will point to the nodes that will be compared. Column has the value of the column (and the reference parameter will transfer the way how to handle it?)

(I am not sure about the translation and what it means so I will leave the original text here.)
"Bei Node1 und Node2 handelt es sich um Zeiger auf die zu vergleichenden Nodes. Column hat den Wert der Spalte und dem Referenzparamter wird die Handlungsanweisung übergeben."

(If node1 should stand in front of node2, the result must be smaller then 0.) If both values are identical, the result must be set to 0, and in case Node2 is bigger then node1 the result will be a value > 1.

(I am again not sure about the translation and what it means so I will leave the original text here.)
Soll Node1 vor Node2 stehen, muss Result einen Wert kleiner als 0 haben. Sind beide Werte identisch muss Result auf 0 gesetzt werden und falls Node2 größer als Node1 ist, bekommt Result einen Wert >1.

8.1 An Example

First we will insert a few nodes to the tree and attach the next record:

```
  PTreeData = ^TTreeData;
  TTreeData = record
    TestStr: String;
  end;
```

The variable TestStr is our compare variable, which we will use also as the caption of the node. How we will do this should be clear because of the earlier chapters.

Set the event-handler for the OnCompare-event en fill it with the following source code:

```
procedure TForm1.vstCompareNodes(Sender: TBaseVirtualTree; Node1,
  Node2: PVirtualNode; Column: Integer; var Result: Integer);
var
  Data1: PTreeData;
  Data2: PTreeData;
begin
  Data1:=vst.GetNodeData(Node1);
  Data2:=vst.GetNodeData(Node2);

  if (not Assigned(Data1)) or (not Assigned(Data2)) then
    Result:=0
  else
    Result:=CompareText(Data1.TestStr, Data2.TestStr);
end;
```

What is happening here? First the data, of the two nodes that have to be compared, is retrieved. This is done by the method GetNodeData. We need this data, to get our other compare variable Teststr. The if-statement will test if the data is available. If it is not the program will crash sooner or later.

The last line is important. The routine CompareText compares two strings and conveniently returns the right variables (>0, when the first value is bigger then the second; 0, if both values are identical, and < 0 in all the other cases). The routine requires two strings. In our case Data1.TestStr (of the first node) and Data2.TestStr (of the second node).

Now we have to give the tree the command to sort its nodes. We will do this by the routine SortTree. The column that was clicked on is set in the parameter Column, which will hold the Index-number of the column. So you could also use a case-statement to react on every column. In our example the strings TestStr are always compared, no matter what column you clicked on.

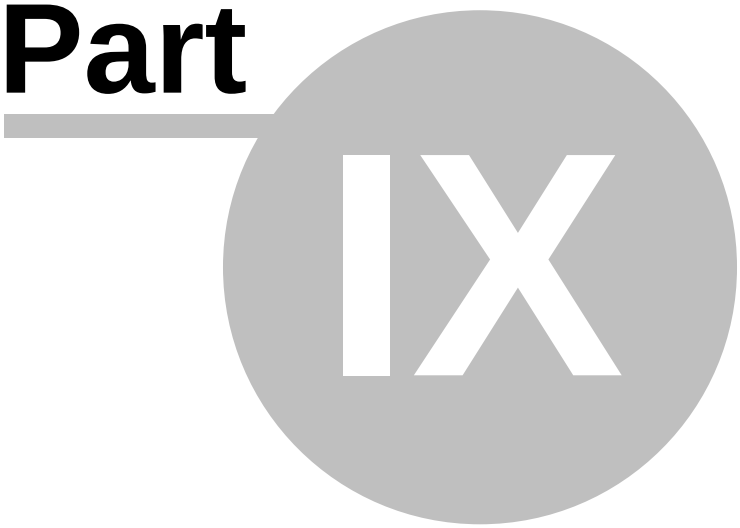
A complete example is downloaded here.

Often columns are sorted by clicking on them. To realise that a little modification to the OnHeaderClick-event has to be made:

```
procedure TMainForm.vstThreadsHeaderClick(Sender: TVTHeader;
  Column: TColumnIndex; Button: TMouseButton; Shift: TShiftState; X,
  Y: Integer);
begin
  vst.SortTree(Column,Sender.SortDirection,True);
  if Sender.SortDirection=sdAscending then
    Sender.SortDirection:=sdDescending
  else
    Sender.SortDirection:=sdAscending
end;
```

The property SortDirection of the header line determines the sorting direction, which most be toggled on every click.

Use icons and images



9 Use icons and images

To make a node stand out, you can place an small icon next to it. The image that has to be shown must be loaded in an ImageList. This can be done in design-time. Now the image list must be attached to the tree, do this by using the property Images and pick the list.

The event OnGetImageIndex will be handling the showing of the icons:

```
procedure TForm1.vstGetImageIndex(Sender: TBaseVirtualTree;
  Node: PVirtualNode; Kind: TVTImageKind; Column: Integer;
  var Ghosted: Boolean; var ImageIndex: Integer);
```

As you can see, the parameter list is relatively long. The parameter 'Sender' holds the component from which the call came. 'Node' points to the node in question and 'Column' returns the current column. The parameter will have the value -1 if these are no columns defined.

The meaning of the parameters we just discussed should be clear, by reading the other chapters. New are the parameters 'Kind' and the reference parameter 'Ghosted' and 'ImageIndex'.".

'Kind' is an enumerated list with the following structure:

```
TVTImageKind = (
    ikNormal,
    ikSelected,
    ikState,
    ikOverlay
  );
```

If the status of the node is normal, then 'Kind' has the value ikNormal. If the node is for example selected it has the value 'ikSelected'.

If you set the Boolean parameter 'Ghosted' to true, the icon will be disabled. The default value is false.

ImageIndex must be set to a value in the ImageList, to the picture that will be shown. The count starts at 0.

An example could look like this. For this example you will need a tree with at least two columns and an ImageList with two icons.

```
procedure TForm1.vstGetImageIndex(Sender: TBaseVirtualTree;
  Node: PVirtualNode; Kind: TVTImageKind; Column: Integer;
  var Ghosted: Boolean; var ImageIndex: Integer);
begin
  case Kind of
    ikNormal, ikSelected:
      case Column of
        0: ImageIndex:=0;
        1: if Sender.FocusedNode = Node then
             ImageIndex:=1;
      end;
  end;
end;
```

In this example you will see two case-statements, one nested in th other. The first case-condition is only met when the node is either selected or normal. In this example an ImageList will be shown next to the first column. If a node has focus then the second image is shown in this second column.

A complete example can be downloaded here.

9.1 Choose your own background

By using the property Background in the tree it is possible to select your own background image. To ensure the image is shown you must set the value toShowBackground in the property TreeOptions|PaintOptions to true. The image must be a bitmap. The property BackgroundOffsetX and BackgroundOffsetY define how many pixels the image must be moved from the top-left angle. The image will be tilled automatically.

9.2 Fonts for the node

The fonts can be set for every node. This is done in the event OnPaintText of the tree. This event is comparable with the OnDrawItem-event of the TListbox, in which you also can influence the Paint progress. The procedure header of the event-handler look like this:

```
procedure TForm1.vstPaintText(Sender: TBaseVirtualTree;
  const TargetCanvas: TCanvas; Node: PVirtualNode; Column: Integer;
  TextType: TVSTTextType);
```

Interesting here is the parameter TargetCanvas of the type TCanvas. It is declared as a constant, but this is not really important because with an object you only pass a reference. (By this parameter the fonts will be drawn.?) ("Über diesem Parameter werden die Schriftformatierungen vorgenommen."). All the possibilities of the TCanvas object are at your disposal. Node hold the node in question and column the column.

The procedure could look like this. In this example the tree must at least have two visible columns.

```
procedure TForm1.vstPaintText(Sender: TBaseVirtualTree;
  const TargetCanvas: TCanvas; Node: PVirtualNode; Column: Integer;
  TextType: TVSTTextType);
begin
  with TargetCanvas do
    case Column of
      0: Font.Style:=Font.Style + [fsBold];
      1: Font.Color:=clRed;
    end;
end;
```

All the nodes of the first column (column 0) will be set to bold, the nodes in the second column will get the color red.

Save and Load

Part X

10 Save and Load

The saving and loading of a tree is easier then you might think, when you first look at it. Although it is more complex then it was with the TTreeView component, it can still be done with just a few lines.

The tree provides the methods SaveToFile and LoadFromFile, which will just need a (full) filename as parameter. These methods will save build-up and the structure of the node (including information if a node is collapsed or not) of the tree.

Saving:

```
procedure TForm1.Button2Click(Sender: TObject);
begin
  vst.SaveToFile('C:\VirtualTree1.dat');
end;
```

Loading:

```
procedure TForm1.Button3Click(Sender: TObject);
begin
  vst.LoadFromFile('C:\VirtualTree1.dat');
end;
```

Most of the time there is much more data that is attached to each node. This data is not loaded nor saved by the methods we just mentioned. To simplify the saving and loading of extra data the tree has provided the OnSaveNode- and OnLoadNode- events:

```
procedure TForm1.vstSaveNode(Sender: TBaseVirtualTree; Node: PVirtualNode;
  Stream: TStream);
```

Those events have the same procedure-header. By now you will understand what the parameters Sender and Node are for. For the first time we see the parameter Stream. Stream is the stream the data is added to. Because the stream is progressing (a node is attached to another node), it will be enough to just simply writing the data directly in the stream by using Write. You don't have to worry about the memory reservation or the releasing of the Stream.

We assume that our data-record has the following structure:

```
type
  PTreeData = ^TTreeData;
  TTreeData = record
    TestStr: String
  end;
```

10.1 Saving

To save this data, you could do this (event: OnSaveNode):

```
procedure TForm1.vstSaveNode(Sender: TBaseVirtualTree; Node:
PVirtualNode;
  Stream: TStream);
var
  Data: PTreeData;
  Len: integer;
begin
  Data := vst.GetNodeData(Node);
  Len := Length(Data.TestStr);
  Stream.write(Len, SizeOf(Len));
  Stream.write(PChar(Data.TestStr)^, Len);
end;
```

By using GetNodeData you can retrieve the address of the data. In our case the record is just one string. Because the length of the string is dynamic, the actual length must be saved, otherwise we will get into trouble when loading the data. The length in bytes will be saved as the Integer-variable Len and written in the stream by using Stream.Write. Directly after this the string will be written. The first parameter will b the address of the pointer, then the length in Bytes. The length/size of the data types is mostly retrieved by using SizeOf (like with integer-types) or Length (like in this case with a string).

If your data contains an integer type, save it like this:

```
  Stream.Write(Data.IntegerVar, SizeOf(Data.IntegerVar));
```

(When you save a tree like this it is recommended to set the property TreeOptions|StringOptions so that the value 'toSaveCaptions' is on?) (or is it off?)

("Wenn du deinen Baum auf diese Weise abspeicherst, empfiehlt es sich die Eigenschaft TreeOptions.StringOptions so zu bearbeiten, dass der Wert 'toSaveCaptions' nicht enthalten ist.")

10.2 Loading

The loading of the tree structure is barely different from saving the structure. This time we find ourselves in the OnLoadNode-event:

```
procedure TForm1.vstLoadNode(Sender: TBaseVirtualTree; Node:
PVirtualNode;
  Stream: TStream);
var
  Data: PTreeData;
  Len: integer;
begin
  Data := vst.GetNodeData(Node);
  Stream.read(Len, SizeOf(Len));
  SetLength(Data.TestStr, Len);
  Stream.read(PChar(Data.TestStr)^, Len);
end;
```

First you must get the memory address van the data. After that the long string which we stored as integer is read. With SetLength we specifically set the size in the memory, which we have already retrieved with SizeOf. Finally we will use Stream.Read to read out our strings. The function will need a start address of the string (PChar(Data.TestStr)^) and the length (Len).

Remember to read out the data in the exact order in which you have stored it.

Now the tree should save the entire tree with data and with LoadFromFile ehe entire tree should be loaded.

An example of this chapter can be downloaded here. The example is bases on the demo's you will get with the VirtualTree-Archives.

10.3 Make editing possible

The next chapter will show you how to make it possible for the users to edit a node on a later time. This functionality you will know from the Windows Explorer, where you rename data like this.

The component provides four event that handle editing. OnEditing is called when the tree is on the verge of giving the giving the user access to the editing field. There is still time to prohibit editing at this moment. Is a node successfully edited, then the OnNewText-event is called. This is the time the new text is assigned to the node. Then the event OnEdited will come next which will finish a successfully edited field. If the editing is cancelled (by for instance the Esc button), the event EditCancelled is called.

Before you can even start by edit nodes, you have to add the value 'toEditable' to TreeOptions|MiscOptions. Is the property not set, user editing will be impossible. Now it should be possible to edit the main column (normally the column next to the left edge).

In our example we will set up the following data-record:

```
type
  PTreeData = ^TTreeData;
  TTreeData = record
    TestStr: String
  end;
```

TestStr will be the caption of the node.

We want editing to be possible if the node is the root node. So the function GetNodeLevel should return 0. The event to create this statement should be OnEditing:

```
procedure TForm1.vstEditing(Sender: TBaseVirtualTree;
  Node: PVirtualNode; Column: TColumnIndex; var Allowed: Boolean);
begin
  Allowed:=Sender.GetNodeLevel(Node)<1;
end;
```

The parameter Allowed, if true, will make editing possible. Node is the pointer set to the node that is set to be edited.

If the user has entered a text en confirmed with 'return', then the event OnNewText is called.
This is where the new text is stored into the data-record:

```
procedure TForm1.vstGraphicsNewText(Sender: TBaseVirtualTree;
  Node: PVirtualNode; Column: TColumnIndex; NewText: WideString);
var
  Data: PTreeData;
begin
  Data:=Sender.GetNodeData(Node.Parent);
  if Assigned(Data) then
    Data.Description:=NewText;
end;
```

The parameter NewText holds the new text that the user has just entered.

(With the property delay you set the delay, in milliseconds, that the Edit will be polled, drawn or activated?) The default value is 1000.
Über die Eigenschaft EditDelay steuern Sie, mit welcher Verzögerung in Millisekunden das Editier-Feld angezeigt werden soll. Standardmäßig steht die Eigenschaft auf dem Wert 1000.