Virtual Treeview

Компонент отображения деревьев и списков: Virtual Treeview.
Приёмы использования в проектах.
http://www.soft-gems.net

Последняя версия статьи и исходные коды всех проектов, представленных в материалах статьи, всегда могут быть скачаны по адресу:
http://quadr0.pochta.ru/VirtualTreeview/virtualtreeview.zip

Версия статьи в Adobe PDF формате доступна по адресу:
http://quadr0.pochta.ru/VirtualTreeview/virtualtreeview_pdf.zip

Со мной можно связаться:
E-mail: 4quadr0@qmail.com

Кроме того, поймать меня можно на форуме программистов «Vingrad»:
http://www.vingrad.ru
http://vingrad.ru/@Quadr0

Содержание

1.0. Введение

Всё чаще стали встречаться вопросы по этому, безусловно, отличному компоненту, и всё больше программистов переписывают свои проекты для использования с VT (здесь и далее Я буду применять сокращение «VT», вместо «Virtual Treeview»). Соответственно назрела идея написания этой статьи. Да и не просто назрела, а стала необходимостью, так как существующий по компоненту официальный документ не приводит толком никакого кода, ограничиваясь лишь пустым описанием возможностей и иерархий классов. В этой статье Я хотел бы осветить следующее:

- Назначение компонента;
- Его преимущества или недостатки перед стандартными TTreeView и TListView (да, да, несмотря на своё название, VT является помесью и того и другого одновременно);
- Приёмы и примеры использования компонента в Ваших проектах;
- А также предостеречь Вас о возможных ошибках при неправильном использовании компонента.

Сразу хочу сказать, что компонент поначалу может показаться Вам слишком громоздким и тяжелым. Но уверяю Вас, как только Вы сделаете первые шаги и начнете работать с этим компонентом, Вы долго потом будете удивляться, почему сразу всё не было реализовано именно так, как это сделано в VT.

1.1. TVirtualTreeView - проект SoftGems (http://www.soft-gems.net).

История компонента начинается в дремучем 1999 году, когда его автор и по существу единственный разработчик вообще, **Mike Lischke**, после релиза TreeNT (прародителя VT) устроился на новое место работы. Внедряя TreeNT в корпоративную библиотеку компании, он впервые осознал как велико различие между требованиями к индивидуальному, профессиональному программному обеспечению и корпоративным разработкам. TreeNT был враппером вокруг Microsoft'ского CommCtl32.dll компонента (TTreeView) и потому нёс в себе все самые существенные его недостатки, а именно: низкую скорость работы (добавление 5000-6000 узлов занимало несколько минут), сильную зависимость от версий библиотеки и отсутствие кроссплатформенности.

Было принято решение написать компонент отображения дерева «с нуля» на чистом VCL, который был бы ориентирован на работу с виртуальными данными. Напомню, что это принцип, при котором дерево абсолютно ничего не знало бы о данных своих узлов, даже их текст, например. Единственное, о чём знает VT это структура дерева, представленная, грубо говоря, связанным списком, где каждый элемент – запись (TVirtualNode), хранящая служебную информацию узла (настройки, состояние, индекс и т.д.). О том, как дерево управляется с данными, будет сказано позже.

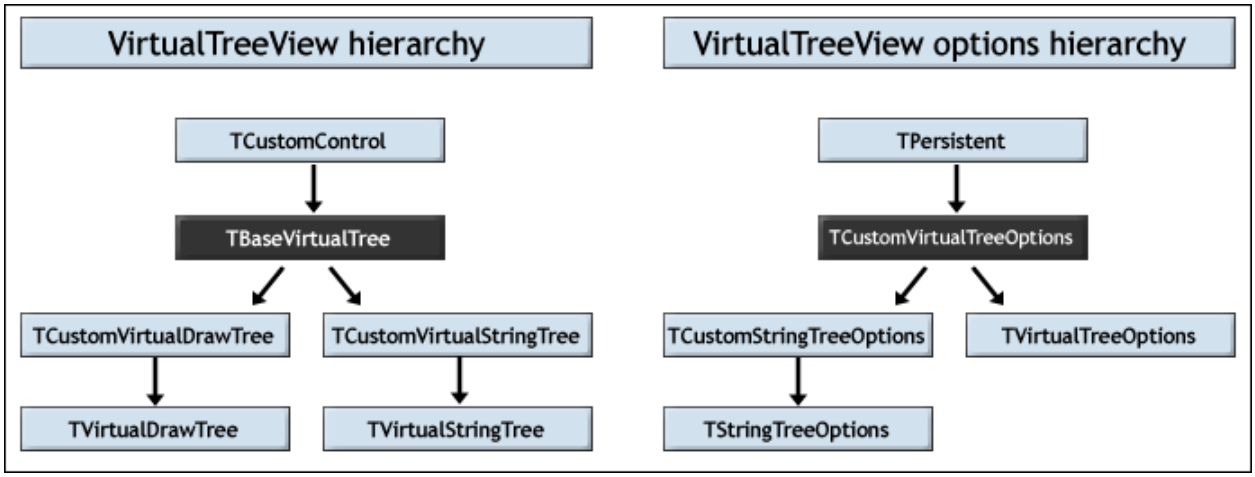
Текущая версия компонента (4.3 на момент написания этих строк) реализует прорву возможностей. Перечислять Я их здесь, понятное дело, не буду – этот список займёт около двух-трёх страниц печатного текста. Скажу только, что всё, ВСЁ, что Вы могли бы захотеть сделать с деревом, Вы сделаете с VT. Думаю, для Вас также приятно будет узнать, что VT используется в средах Borland, начиная с BDS 3 (Delphi 2005), что можно считать гарантом его безупречной стабильности в работе.

1.2. Применение в проектах

Всё бы хорошо, но какая польза может быть от этого большого, как уж ни крути, компонента, скажем, для вашей программы-базы данных телефонной книжки или миниатюрного просмотрщика папок, где количество узлов вряд ли когда-нибудь превысит сотню? На самом деле всякая :). Помимо своих скоростных преимуществ, VT позволяет контролировать каждый, даже самый маленький механизм своей работы. Около ста (!) событий позволят Вам изменить отрисовку дерева (например, добавить градиентную заливку), увеличить высоту узлов, цвет и стиль текста, добавить всплывающую подсказку для обрезающегося длинного текста или нарисовать подсказку самому, с помощью Ваших компонентов и методов. Также можно самому отрисовывать колонки, заголовок каждой колонки, отслеживать малейшие изменения дерева, определять порядок сортировки, добавлять собственные редакторы в дерево (кроме обычного текстового поля вроде TEdit) и т.д. Словом, Я Вас уверяю, даже самая неприхотливая программка, работающая со списками, тут же преобразится от использования VT, прибавив в **профессиональности**, **интерактивности** и **технологичности**.

1.3. Иерархия классов.

Прежде чем мы начнём, взгляните на эту схему, чтобы ознакомиться с «внутренностями» VT.



Как видите, «Virtual Treeview» это всего лишь общее название для двух классов: TVirtualDrawTree и TVirtualStringTree, которые, в свою очередь, являются потомками базового для всех деревьев класса – TBaseVirtualTree.

2.0. Инициализация данных

Достаточно истории и дифирамб, давайте приступим ближе к сабжу и напишем своё первое приложение с VT. Скачивайте пакет для своей версии Delphi с http://www.soft-gems.net/VirtualTreeview/ и приготовьтесь к нашим увлекательным экспериментам :).

2.1. Базовые операции. Создание колонок. Управление контентом дерева. Инициализация данных для дерева.

В этой главе нашей с Вами задачей будет разобраться в том, как VT управляет своими виртуальными данными, а также научиться грамотно заполнять дерево и работать с его структурой. Здесь же будут рассмотрены обработчики самых базовых событий компонента, без которых его работа невозможна.

Итак, Вы кинули компонент на форму. Перво-наперво, неплохо было бы заполнить дерево какими-нибудь узлами, подумаете Вы. Но не тут-то было. В этом заключается первое неудобство компонента - все данные загружаются в дерево исключительно и только в рантайме. Тем не менее, предусмотрено свойство **RootNodeCount**, настраиваемое из Object Inspector, которое позволяет добавить в дерево n-ое количество корневых элементов.

Запомните: самым корневым элементом, родителем всех веток является **RootNode**. Это следует учитывать при рекурсивных проходах дерева. Соответственно у RootNode в поле Parent будет значиться уже **nil,** и в самом дереве она видна естественно не будет.

При изменении этого свойства дерево очищается и RootNode заполняется дочерними элементами. В любом случае, забивать данными эти узлы нам всё равно придётся в рантайме :).

Как уже было сказано выше, физически «заполнить» VT данными невозможно. Для работы с ними компонент использует мощный событийный механизм. Т.е. все данные, которые Вы хотите отобразить в дереве, Вы будете отображать с помощью событий отрисовки, получения текста и т.д. Рассмотрим пример.

У нас есть некая простая база данных телефонной книжки, представленная двумя массивами:

```
Names: array[0..4] of WideString = (
  'Вася',
  'Петя',
  'Маша',
  'Костя',
  'Дима'
);

Phones: array[0..4] of WideString = (
  '433-56-49',
  '545-67-79',
  '777-50-50',
  '911-03-05',
  '02'
);
```

Как Вы уже, наверное, заметили, нам потребуется 2 колонки для отображения данных в нашей VT-таблице. Добавление колонок осуществляется через класс **TVTHeader** и его поле **Columns**, обитающий в поле «**Header**» у VT. Самое время сделать паузу и рассмотреть поближе наиболее интересные настройки и свойства этого класса, назначение которого - управлять колонками. Я буду уделять внимание только самым существенным настройкам, думаю, Вы и сами знаете, зачем компоненту могут понадобиться свойства наподобие **Color**, **ImageIndex** или **Text**.

- **AutoSizeIndex** - позволяет Вам указать какая из колонок будет иметь приоритет для авто-подбора ширины (грубо говоря, какая колонка будет растягиваться при изменении ширины дерева). Работает только вместе с включённой опцией **hoAutoResize**.

- **Columns** – непосредственно редактор колонок. Каждая колонка представляет собой экземпляр класса TVirtualTreeColumn в коллекции. Обратим на него внимание:
  - **Layout** – определяет положение глифа (картинки) на заголовке колонки.
  - **Margin** – задаёт отступ для текста заголовка.
  - **MaxWidth, MinWidth** – максимально и минимально допустимая ширина колонки соответсвенно.
  - **Options** – индивидуальные настройки колонки, они имеют больший приоритет, чем настройки TVTHeader, т.е. могут активировать или выключать какие-то свойства для данной конкретной колонки, независимо от настроек всего заголовка.
    - **coAllowClick** – аналог свойства HotTrack для TListView и его колонок. При наведении курсора на заголовок колонки, он будет изменять свой вид.
    - **coDraggable** – определяет можно ли таскать колонку мышью по заголовку, меняя её положение.
    - **coEnabled**, **coParentBiDiMode**, **coParentColor**, **coResizeable** – определяют состояние, направление текста (для арабских языков, напр.), цвет колонки, а также можно ли изменять ширину колонки.
    - **coShowDropMark** – определяет будет ли на заголовке колонки отрисовываться небольшой значок, показывающий направление вставки перетаскиваемой колонки.
    - **coAutoSpring** – очень интересное свойство. Оно определяет будет ли колонка поддерживать свойство **hoAutoSpring** у TVTHeader. Назначение этих свойств будет рассмотрено чуть ниже.
    - **coFixed** – если это свойство включено, то данная колонка будет полностью заблокирована (прокрутка, доступ к узлам и т.д.) и не будет отвечать на запросы пользователя. Очень полезно при проектировании многопотоковых приложений, когда Вы хотите на определённый момент времени запретить пользователю вмешиваться в содержимое этой колонки.
  - **Position** – положение колонки в заголовке.
  - **Style** – позволяет Вам указать будет ли колонка отрисовываться по умолчанию (**vsText**) или Вы хотите переопределить её процедуру отрисовки и всё делать сами (**vsOwnerDraw**).
- **MainColumn** – определяет индекс основной колонки. Именно элементы этой колонки будут выделяться синим цветом и получать фокус при выделении, и именно напротив элементов этой колонки будут располагаться отметки и другие дополнительные элементы управления дерева.
- **Options** – настройки заголовка. Вот тут начинается самое интересное. :)
  - **hoAutoResize** – если включено, то колонка с индексом из свойства AutoResizeIndex будет занимать всё доступное пространство дерева при изменении его ширины. Ширина такой колонки будет равна: ширина дерева – ширина всех других колонок. Во всём дереве, понятное дело, может быть только одна такая колонка.
  - **hoColumnResize** – Глобальное свойство, определяющее можно ли менять ширину колонок.
  - **hoDblClickResize** – при двойном клике на данной колонке она примет ширину самого длинного её элемента. Рекомендуется держать в True. Опция подавляется свойством hoAutoResize.
  - **hoDrag** – определяет можно ли перетаскивать колонки.
  - **hoRestrictDrag** – Не даёт курсору тянуть заголовок перетаскиваемой колонки к низу или вверх.
  - **hoShowSortGlyphs** – определяет будет ли на заголовке колонки отрисовываться небольшой значок, показывающий текущее направление сортировки.
  - **hoAutoSpring** – вот то свойство, о котором нужно сказать отдельно. Оно позволяет колонкам с включённым свойством coAutoSpring пропорционально изменять свою ширину при изменении ширины дерева. Учтите, однако, что это свойство подавляется свойством hoAutoResize.
- **SortColumn** – индекс колонки, относительно содержания которой сортируется дерево.
- **SortDirection** – направление сортировки колонки с индексом **SortColumn.**
- **Style** – позволяет изменить внешний вид заголовка. Активно только при выключенных темах оформления Windows XP.

Добавьте две колонки и выставьте настройки по своему вкусу. Я установил свойство AutoSpring у всех элементов в True, сделал заголовок видимым и добавил ему немного высоты. Теперь нам во всеоружии можно вернуться к примеру с телефонной книжкой, чтобы грамотно наполнить дерево контентом.

Для добавления веток в дерево у VT существует два базовых метода. Это **AddChild**:

```
function TBaseVirtualTree.AddChild(Parent: PVirtualNode; UserData: Pointer =
nil): PVirtualNode;
```

И **InsertNode**:

```
function TBaseVirtualTree.InsertNode(Node: PVirtualNode; Mode:
TVTNodeAttachMode; UserData: Pointer = nil): PVirtualNode;
```

Оба они возвращают указатель на добавленный узел. Последний отличается от первого лишь тем, что позволяет указать позицию добавляемого узла в свойстве **Mode**:

- **amNoWhere** - узел не добавляется вообще.
- **amInsertBefore** - узел добавляется на позицию выше по отношению к Node.
- **amInsertAfter** - узел добавляется на позицию ниже по отношению к Node.
- **amAddChildFirst** - узел добавляется первым дочерним элементом в Node.
- **amAddChildLast** - узел добавляется последним дочерним элементом в Node.

В нашем первом примере мы коснёмся лишь первого метода. Заполним, наконец, наше дерево:

```
procedure TfrmMain.BtnLoadClick(Sender: TObject);
var
  i: Integer;
begin
  for i := 0 to Length(Names) - 1 do
    VT.AddChild(nil);
end;
```

Здесь мы просто добавили количество веток, равное количеству имён в телефонной книжке. Но большой пользы от пустых узлов не добьёшься. Синхронизируем наше дерево с данными. В частности, нам необходимо отобразить текст из телефонной книжки. «Заполнение» дерева текстом осуществляется через событие **OnGetText**:

```
procedure TfrmMain.VTGetText(Sender: TBaseVirtualTree; Node: PVirtualNode;
  Column: TColumnIndex; TextType: TVSTTextType; var CellText: WideString);
begin
  case Column of
    // Используем остаток от деления. Массив может неожиданно кончиться
    // после нескольких нажатий BtnLoad подряд без очистки.
    // Спасибо OverLord за это исправление (the__teacher@mail.ru).
    // Текст для колонки имени
    0: CellText := Names[(Node.Index) mod (Length(Names))];
    // Текст для колонки телефонного номера
    1: CellText := Phones[(Node.Index) mod (Length(Phones))];
  end;
end;
```

Первый параметр события содержит ссылку на экземпляр дерева, вызвавшего это событие, второй - узел, для которого получаем текст, третий - индекс колонки, для которой берётся текст, четвёртый - тип текста, будет рассмотрен позже, и, наконец, пятый - строка, которую мы и должны наполнить данными. В нашем примере мы просто берём текст из двух массивов для соответствующих колонок.

Теперь, когда мы реализовали обработку этого события, мы также можем получать текст посредством метода **Text**, который в свою очередь вызывает событие OnGetText для получения данных. Сделано это для Вашего удобства, чтобы не писать сотню раз код извлечения данных из узла.

```
property Text[Node: PVirtualNode; Column: TColumnIndex]: WideString;
```

Вы наверняка заинтересовались, а что такое CellText? Ведь дереву не нужны никакие данные, следственно и передавать ничего не надо. Правильно, этот параметр нужен дереву только при перерисовке дерева или при вызове вышеупомянутого **Text**. Он нигде не хранится и заполняется заново при каждом событии отрисовки.

Проект Figure 1.0, демонстрирующий инициализацию дерева, находится в папке Fig 1.0.

2.2. Углублённая работа с данными.

Мы рассмотрели довольно простой пример. Но что делать, если Вы оперируете с динамическими данными? Хранить их в динамическом массиве – затея на грани невозможности по многим причинам. А Вам при этом требуется удалять и добавлять ветки, изменять их содержимое и т.д.?

К счастью, в компоненте предусмотрена возможность хранения некоего указателя на данные для каждого узла, в области памяти которого мы сможем разместить любые нужные нам данные для каждой ветки.

Рассмотрим следующий пример.

Создадим универсальную телефонную книжку, которую динамически можно было бы изменять.

Для начала, решим, как на этот раз мы будем хранить данные каждой ветки. Вариант с массивами исключён, так что же тогда? Для этого воспользуемся записями (record), которые и будут храниться в указателе на данные для каждого узла:

```
type
  PPhoneNode = ^TPhoneNode;

  TPhoneNode = record
    Name, // Имя контакта
    Phone: WideString; // Телефон
  end;
```

Но прежде чем мы напишем код для добавления новых веток, Я хотел бы обратить Ваше внимание на важнейшую деталь. Поскольку VT возвращает указатель на данные, он должен как-то определять положение этих данных в памяти, чтобы мы вдруг не получили **nil** или вообще другие данные, к дереву никак не относящиеся. Для этого в дереве предусмотрено свойство **NodeDataSize**, обозначающее размер данных для каждого узла. С помощью этого свойства дерево способно находить именно данные определённого узла посредством метода **GetNodeData**:

```
function TBaseVirtualTree.GetNodeData(Node: PVirtualNode): Pointer;
```

Разработчик обязан заполнить это свойство, если он собирается использовать такой подход работы с данными, иначе компонент сообщит об ошибке инициализации.

Делается это предельно просто, с помощью стандартной Дельфийской функции **SizeOf**, возвращающей размер блока памяти для определённого типа данных. Делать это лучше сразу после создания формы в обработчике **OnCreate**:

```
procedure TfrmMain.FormCreate(Sender: TObject);
begin
  VT.NodeDataSize := SizeOf(TPhoneNode);
end;
```

В нашем случае размер данных будет равен 8 байтам (по четыре на каждый WideString).

Теперь можно смело заполнять наше дерево узлами:

```
procedure TfrmMain.BtnAddClick(Sender: TObject);
var
  NewNode: PVirtualNode;
  NewPhone: PPhoneNode;
begin
  NewNode := VT.AddChild(VT.FocusedNode);
  NewPhone := VT.GetNodeData(NewNode);
  if Assigned(NewPhone) then
    with NewPhone^ do
    begin
      Name := EdName.Text;
      Phone := EdPhone.Text;
    end;
end;
```

Всё просто: мы добавлям ветку, получаем указатель на область её данных в памяти и заполняем эти самые данные.

Позаботимся и об удалении существующих узлов. Так можно удалить узел, имеющий на данный момент фокус:

```
procedure TfrmMain.mEntryClick(Sender: TObject);
begin
  VT.DeleteNode(VT.FocusedNode);
end;
```

Есть также способ удалить лишь дочерние элементы узла, оставив его самого:

```
procedure TfrmMain.mChildrenClick(Sender: TObject);
begin
  VT.DeleteChildren(VT.FocusedNode);
end;
```

Рассмотрим теперь и вставку новых элементов с помощью метода **InsertNode**:

```
//---------------------------------------------------------------------------
// Вставляем новый элемент перед активным узлом
//---------------------------------------------------------------------------
procedure TfrmMain.BtnInsertBeforeClick(Sender: TObject);
var
  NewNode: PVirtualNode;
  NewPhone: PPhoneNode;
begin
  NewNode := VT.InsertNode(VT.FocusedNode, amInsertBefore);
  NewPhone := VT.GetNodeData(NewNode);
  if Assigned(NewPhone) then
```

```
    with NewPhone^ do
    begin
      Name := EdName.Text;
      Phone := EdPhone.Text;
    end;
end;

//---------------------------------------------------------------------------
// Вставляем новый элемент после активного узла
//---------------------------------------------------------------------------
procedure TfrmMain.BtnInsertAfterClick(Sender: TObject);
var
  NewNode: PVirtualNode;
  NewPhone: PPhoneNode;
begin
  NewNode := VT.InsertNode(VT.FocusedNode, amInsertAfter);
  NewPhone := VT.GetNodeData(NewNode);
  if Assigned(NewPhone) then
    with NewPhone^ do
    begin
      Name := EdName.Text;
      Phone := EdPhone.Text;
    end;
end;

//---------------------------------------------------------------------------
// Вставляем новый элемент первым дочерним элементом активного узла
//---------------------------------------------------------------------------
procedure TfrmMain.BtnChildFirstClick(Sender: TObject);
var
  NewNode: PVirtualNode;
  NewPhone: PPhoneNode;
begin
  NewNode := VT.InsertNode(VT.FocusedNode, amAddChildFirst);
  NewPhone := VT.GetNodeData(NewNode);
  if Assigned(NewPhone) then
    with NewPhone^ do
    begin
      Name := EdName.Text;
      Phone := EdPhone.Text;
    end;
end;

//---------------------------------------------------------------------------
// Вставляем новый элемент последним дочерним элементом активного узла
//---------------------------------------------------------------------------
procedure TfrmMain.BtnChildLastClick(Sender: TObject);
var
  NewNode: PVirtualNode;
  NewPhone: PPhoneNode;
begin
  NewNode := VT.InsertNode(VT.FocusedNode, amAddChildLast);
  NewPhone := VT.GetNodeData(NewNode);
  if Assigned(NewPhone) then
    with NewPhone^ do
    begin
      Name := EdName.Text;
      Phone := EdPhone.Text;
    end;
end;
```

Я специально не стал разделять процедуры на блоки для большей наглядности, но за Вами остаётся отдать дань полиморфизму, выделив повторяющиеся блоки кода в отдельные методы. :)

Проект Figure 1.1, демонстрирующий работу с данными в VT, заполнение дерева и редактирование лежит в папке Fig 1.1.

2.3. Обзор основных событий компонента.

Дерево из простых надписей не очень-то радует глаз пользователя. Давайте добавим интерактивности в наши программы. В частности реализуем картинки для узлов, статический текст, редактирование узлов встроенным редактором VT, сортировку, поиск по инкременту и перебор всех элементов дерева.

Продолжая нашу тему телефонной книжки, для всего этого нам потребуется запись примерно такого содержания:

```
type
  PPhoneNode = ^TPhoneNode;

  TPhoneNode = record
    Name, // Первая колонка
    Mobile, // Вторая колонка
    HomePhone: WideString; // Это будет статический текст второй колонки
    Enabled, // Для отображения состояния выключенности
    Editable: Boolean; // Можно ли редактировать узел
    ImageIndex: Integer; // Индекс картинки в TImageList
    Fg, Bg: TColor; // Цвет текста и фона соответсвенно
  end;
```

2.3.1. Отображение картинок рядом с узлами.

Начнём с картинок. Представим, что мы хотим позволить пользователю выбрать пол контакта и при этом менять картинку на более тёмную, как это делает IE при наведении на неё курсором. Нам понадобится три TImageList, чтобы на их примере показать как в VT можно сделать отображение 3 состояний картинки (нормальная, активная, выключенная). Если Вам лень или просто негде взять таковые, то можете забрать их из моего примера Fig 1.2, скачав аттач в конце статьи.

По идее, картинки к VT подключаются через всем известное свойство **Images**. Однако Я в процессе работы с компонентом часто замечал странные глюки картинок, подключенных таким методом. В компоненте помимо этого есть также свойства StateImages и CheckImages. Ни в хелпе, ни в исходниках не даётся никакой существенной информации по всем этим трём параметрам, поэтому Я не могу точно сказать в каких случаях какое свойство использовать. Просто запомните, если Вы заметите какие-либо странности в отрисовке картинок, просто поменяйте используемое свойство. В данном примере Я использовал свойство StateImages.

Получить картинку в VT можно с помощью двух почти одинаковых событий:

- **OnGetImageIndex**;
- **OnGetImageIndexEx**;

Если Вы посмотрите на их входные параметры, то заметите, что отличаются они лишь на один параметр: второе событие позволяет указывать экземпляр TImageList, из которого будет использоваться картинка. В нашем случае это идеальный вариант:

```
procedure TfrmMain.VTGetImageIndexEx(Sender: TBaseVirtualTree;
  Node: PVirtualNode; Kind: TVTImageKind; Column: TColumnIndex;
  var Ghosted: Boolean; var ImageIndex: Integer;
  var ImageList: TCustomImageList);
```

```
var
  PhoneNode: PPhoneNode;
begin
  if Column > 0 then
    Exit; // Картинка будет отображаться только в первой колонке
  ImageIndex := -1;
  PhoneNode := VT.GetNodeData(Node);
  if Assigned(PhoneNode) then
  begin
    ImageList := Self.ImageList; // Обычные картинки
    // HotNode - узел, над которым находится курсор,
    // т.е. активный.
    if Node = Sender.HotNode then
      ImageList := ImageListHot; // Активные картинки
    if not PhoneNode^.Enabled then
      ImageList := ImageListDisabled; // Выключенные картинки
    ImageIndex := PhoneNode^.ImageIndex;
  end;
end;
```

Но не забывайте также и про обычное событие. Чаще всего его реализации вполне достаточно.

Чтобы пример работал корректно нужно также включить опцию TreeOptions.PaintOptions.toHotTrack, чтобы VT имел доступ к HotNode узлу. Т.е. к узлу, над которым в данный момент «завис» курсор.

2.3.2. Статический текст.

Рассмотрим теперь статический текст. Данная возможность предусмотрена в VT для отображения статического, то есть нередактируемого обычными средствами текста. Это позволит Вам запретить редактирование такого текста стандартным редактором VT. Тем не менее, он будет отображаться рядом с обычным текстом. Посмотрим на нашу телефонную книжку. Представьте, что необходимо рядом с номером мобильного указать домашний телефон. При этом мобильники некоторые люди меняют довольно часто, а вот домашний телефон иногда остаётся на всю жизнь. Идеальное решение для такой ситуации - использование статического текста. Перепишем событие получения текста:

```
procedure TfrmMain.VTGetText(Sender: TBaseVirtualTree; Node: PVirtualNode;
  Column: TColumnIndex; TextType: TVSTTextType; var CellText: WideString);
var
  PhoneNode: PPhoneNode;
begin
  PhoneNode := VT.GetNodeData(Node);
  if Assigned(PhoneNode) then
    case Column of
      0: // Имя
        if TextType = ttNormal then
          CellText := PhoneNode^.Name;
      1: // Телефон
      case TextType of
        ttNormal: CellText := PhoneNode^.Mobile; // Мобильный
        ttStatic: CellText := '(' + PhoneNode^.HomePhone + ')'; // Домашний
      end;
    end;
end;
```

Не забудьте также разрешить отображение статического текста, включив опцию дерева **toShowStaticText** из TreeOptions.StringOptions, т.к. по умолчанию она отключена.

Также это свойство идеально подойдёт для отображения всяких префиксов или постфиксов, например, для обозначения физических величин рядом с их числовым значением.

2.3.3. Сортировка.

Теперь предположим, что нам необходимо сортировать контакты по алфавиту. Для таких задач в VT имеется событие для сравнения двух узлов и обработчики нажатий для заголовка и колонок. Попробуем всё это реализовать на примере.

Для начала обработаем событие сравнения двух узлов:

```
procedure TfrmMain.VTCompareNodes(Sender: TBaseVirtualTree; Node1,
  Node2: PVirtualNode; Column: TColumnIndex; var Result: Integer);
begin
  Result := WideCompareStr(VT.Text[Node1, Column], VT.Text[Node2, Column]);
end;
```

В нашем случае обработчик очень похож на WinAPI callback функции сортировки. VT требует передачи точно такого же результата сравнения двух элементов:

- 0 - Элементы равны, оба остаются на месте.
- 0 < - Первый элемент больше второго.
- 0 > - Второй элемент больше первого.

В нашем случае всё за нас делает ОС (выбирает наибольшую по алфавиту строку). Таким образом, если Вам нужно будет отсортировать узлы по величине чисел, то обработчик примет примерно такой вид:

```
procedure TfrmMain.VTCompareNodes(Sender: TBaseVirtualTree; Node1,
  Node2: PVirtualNode; Column: TColumnIndex; var Result: Integer);
var
  Data1, Data2: PDataNode;
begin
  Data1 := Sender.GetNodeData(Node1);
  Data2 := Sender.GetNodeData(Node2);
  if Data1^.Value > Data2^.Value then
    Result := 1
  else if Data1^.Value < Data2^.Value then
    Result := -1
  else if Data1^.Value = Data1^.Value then
    Result := 0;
end;
```

Помните также, что Вы не должны заботиться о порядке сортировки. Т.е. VT сам отредактирует результат сравнения в соответствии с порядком сортировки, и Вам не придётся писать тот же самый код дважды, меняя условия проверки на противоположные.

Теперь нам необходимо отсортировать дерево по клику на определённую колонку заголовка и изменить порядок сортировки на противоположный. Грубо говоря, просто посмотрите как это делает Проводник :). Делается это следующим обработчиком:

```
procedure TfrmMain.VTHeaderClick(Sender: TVTHeader; Column: TColumnIndex;
  Button: TMouseButton; Shift: TShiftState; X, Y: Integer);
begin
  if Button = mbLeft then
  begin
    // Меняем индекс сортирующей колонки на индекс колонки,
    // которая была нажата.
    VT.Header.SortColumn := Column;
    // Сортируем всё дерево относительно этой колонки
    // и изменяем порядок сортировки на противоположный
    if VT.Header.SortDirection = sdAscending then
```

```
    begin
      VT.Header.SortDirection := sdDescending;
      VT.SortTree(Column, VT.Header.SortDirection);
    end
    else begin
      VT.Header.SortDirection := sdAscending;
      VT.SortTree(Column, VT.Header.SortDirection);
    end;
  end;
end;
```

2.3.4. Работа со встроенным редактором текста.

Идём дальше. У TTreeView и TListView есть такая полезная возможность, как небольшой встроенный редактор текста для узлов. Естественно VT был бы не VT, если бы не поддерживал и эту возможность. :)

Редактор включается в настройках по адресу: TreeOptions.MiscOptions.toEditable.
После этого он станет доступен также, как и в TTreeView и TListView (клик по узлу и пауза до появления).

С редактором связаны такие события:

- **OnEditing** - Происходит непосредственно **перед** появлением редактора. С помощью этого события можно запетить его появление по тем или иным причинам.
- **OnEdited** - Происходит сразу же после исчезновения редактора.
- **OnNewText** - Происходит, если после исчезновения редактора его (редактора) текст был изменён. С его (события) помощью можно обновить Ваши данные.
- **OnEditCanceled** - Происходит, если редактор был закрыт нажатием «Esc».

В нашем примере мы рассмотрим два наиболее часто применяемых. Рассмотрим синхронизацию данных с новым текстом:

```
procedure TfrmMain.VTNewText(Sender: TBaseVirtualTree; Node: PVirtualNode;
  Column: TColumnIndex; NewText: WideString);
var
  PhoneNode: PPhoneNode;
begin
  if Length(NewText) = 0 then
    Exit;
  PhoneNode := VT.GetNodeData(Node);
  if Assigned(PhoneNode) then
  begin
    case Column of
      0: PhoneNode^.Name := NewText;
      1: PhoneNode^.Mobile := NewText;
    end;
  end;
end;
```

Теперь мы должны разрешить или запретить редактирование в соответствии со значением поля **Editable** нашей записи. Обработчик **onEditing**:

```
procedure TfrmMain.VTEditing(Sender: TBaseVirtualTree; Node: PVirtualNode;
  Column: TColumnIndex; var Allowed: Boolean);
var
  PhoneNode: PPhoneNode;
begin
  Allowed := False;
```

```
  PhoneNode := VT.GetNodeData(Node);
  if Assigned(PhoneNode) then
    Allowed := PhoneNode^.Editable;
end;
```

Не забудьте обратить внимание на параметр **EditDelay**. Он позволяет установить задержку перед началом редактирования узла. Значение по умолчанию – одна секунда.

2.3.5. Поиск по инкременту.

Вы уже наверняка знакомы с этой возможностью по Visual Studio и BDS. Выглядит это примерно следующим образом: Вы постепенно вводите какой-то текст и VT по мере ввода находит узел, текст которого наиболее совпадает с вводимым. В конце-концов, когда Вы закончите ввод, выделенным останется искомый узел или узел, более-менее похожий на него. Такой подход очень полезен, когда Вы не знаете точного названия ветки, а только её часть. В VT вся эта прелесть работает с помощью события **OnIncrementalSearch**:

```
procedure TfrmMain.VTIncrementalSearch(Sender: TBaseVirtualTree;
  Node: PVirtualNode; const SearchText: WideString; var Result: Integer);
var
  PhoneNode: PPhoneNode;
  Len: Integer;
begin
  Result := 0;
  PhoneNode := VT.GetNodeData(Node);
  if Assigned(PhoneNode) then
  begin
    // Используя StrLIComp, мы можем указать длину сравнения.
    // Таким образом, мы сможем найти узлы, совпадающие частично.
    Result := StrLIComp(PAnsiChar(AnsiString(SearchText)),
      PAnsiChar(AnsiString(PhoneNode^.Name)),
       Min(Length(SearchText), Length(PhoneNode^.Name)));
  end;
end;
```

Для работы такого поиска надо также известить VT о том, что мы написали соответствующий обработчик и поиск возможен. Для этого надо присвоить параметру **IncrementalSearch** любое значение, кроме **isNone**:

- **isAll** – поиск во всех узлах.
- **isInitializedOnly** – поиск только в инициализированных узлах. О событии инициализации будет сказано в последующих главах.
- **isVisibleOnly** – поиск только в видимых узлах. О таких узлах тоже будет сказано чуть позже.

Параметр **IncrementalSearchDirection**, судя из названия, отвечает за направление поиска (вверх-вниз).

Очень важен параметр **IncrementalSearchStart**. Он управляет диапазоном поиска. Принимает следующие значения:

- **ssAlwaysStartOver** – при каждом новом нажатии с клавиатуры поиск начинается заново с самого первого элемента дерева. Медленно, но надёжно.
- **ssFocusedNode** – поиск начинается с выделенного элемента. Компромиссный вариант между двумя остальными.
- **ssLastHit** – самый быстрый, но не такой надёжный в плане точности параметр. Поиск начинается всегда с последнего совпадения.

Параметр **IncrementalSearchTimeout** - позволяет задать паузу между двумя нажатиями клавиш. По истечении этой паузы поиск завершается.

Пожалуй, это всё, что можно сказать о поиске по инкременту. Чтобы его протестировать, кликните по VT и начните нажимать разнообразные сочетания клавиш.

2.3.6. Простейшие приёмы отрисовки.

Что Я там ещё обещал? Ах, да! Отрисовка элементов. Вот два примера как можно скрасить чёрно-белые будни VT.

Меняем цвет текста:

```
procedure TfrmMain.VTPaintText(Sender: TBaseVirtualTree;
  const TargetCanvas: TCanvas; Node: PVirtualNode; Column: TColumnIndex;
  TextType: TVSTTextType);
var
  PhoneNode: PPhoneNode;
begin
  PhoneNode := VT.GetNodeData(Node);
  if Assigned(PhoneNode) then
    TargetCanvas.Font.Color := PhoneNode^.Fg;
  if (vsSelected in Node.States) and (Sender.Focused) then
    TargetCanvas.Font.Color := clHighlightText;
end;
```

И фон узла:

```
procedure TfrmMain.VTBeforeCellPaint(Sender: TBaseVirtualTree;
  TargetCanvas: TCanvas; Node: PVirtualNode; Column: TColumnIndex;
  CellRect: TRect);
var
  PhoneNode: PPhoneNode;
begin
  PhoneNode := VT.GetNodeData(Node);
  if Assigned(PhoneNode) then
    with TargetCanvas do
    begin
      Brush.Color := PhoneNode^.Bg;
      FillRect(CellRect);
    end;
end;
```

А вот так можно на фоне нарисовать что-то наподобие Windows XP заголовка:

```
procedure TfrmMain.VTBeforeCellPaint(Sender: TBaseVirtualTree;
  TargetCanvas: TCanvas; Node: PVirtualNode; Column: TColumnIndex;
  CellRect: TRect);
var
  BigGrad, SmallGrad: TRect;
begin
  if Column = 0 then
  begin
    BigGrad := CellRect;
    Dec(BigGrad.Bottom, 2);
    SmallGrad := BigGrad;
    SmallGrad.Top := SmallGrad.Bottom;
    Inc(SmallGrad.Bottom, 2);
    GradFill(TargetCanvas.Handle, BigGrad, clInactiveCaptionText, clWindow,
gkHorz);
    GradFill(TargetCanvas.Handle, SmallGrad, clHighlight, clWindow, gkHorz);
```

```
  end;
end;
```

Процедуру градиентной заливки можно взять из всё того же проекта Figure 1.2. Берите, не стесняйтесь :).

2.3.7. Рекурсивный перебор всех узлов дерева.

Последний пример данной главы демонстрирует перебор всех узлов VT.

```
procedure TfrmMain.BtnFindClick(Sender: TObject);

  function FindNode(ANode: PVirtualNode;
    const APattern: WideString): PVirtualNode;
  var
    NextNode: PVirtualNode;
    PhoneNode: PPhoneNode;
  begin
    Result := nil;
    NextNode := ANode.FirstChild;
    if Assigned(NextNode) then
      repeat
        PhoneNode := VT.GetNodeData(NextNode);
        if Assigned(PhoneNode) then
          if RbName.Checked then
            if PhoneNode^.Name = APattern then
            begin
              Result := NextNode;
              Exit;
            end
            else
          else
            if PhoneNode^.Mobile = APattern then
            begin
              Result := NextNode;
              Exit;
            end;
        // Ищем в дочерних ветках
        Result := FindNode(NextNode, APattern);
        // Нужно добавить проверку для случая, когда
        // был найден нужный элемент, иначе, даже если
        // элемент был найден, то в результате поиска будет
        // возвращен nil, поскольку в начале каждого вызова
        // процедуры стоит  Result := nil;
        // Изменение внес Кривожиха Максим (airmax@fitbmk.cdu.edu.ua),
        // за что ему спасибо.
        if Result = nil then
          // Переходим на соседнюю ветку...
        NextNode := NextNode.NextSibling;
      until
        //...пока не вмажемся лбом в стену :)
        NextNode = nil;
  end;

var
  FoundNode: PVirtualNode;
begin
  if VT.RootNodeCount > 0 then
  begin
    FoundNode := FindNode(VT.RootNode, EdSearch.Text);
    LoadNode(FoundNode);
    if Assigned(FoundNode) then
    begin
      VT.ClearSelection;
      VT.FocusedNode := FoundNode;
      Include(VT.FocusedNode.States, vsSelected);
    end;
  end;
end;
```

Здесь становится заметно ещё одно маленькое неудобство. В связи с тем, что VT работает со связанными списками, Вы не сможете использовать для переборки цикл **for**. Остаются лишь циклы с проверкой наверху (**while**) и внизу (**repeat..until**).

Примечание: Участник Форума Программистов Vingrad - **RA** нашел весьма необычное, но рабочее решение для использования цикла **for**. Передаю ему слово:

```
var
  I: Integer;
  PNode: PVirtualNode;
  Data: utilUnit.PVSTRecord;
begin
   for I := 0 to VST.TotalCount - 1 do
   begin
     if I <> 0 then
       PNode := VST.GetNext(PNode)
     else
       PNode := VST.GetFirst;
     Data := VST.GetNodeData(PNode);
     // Делаю что хочу :).
   end;
end;
```

2.3.8. Функции, предоставляемые VT для навигации по дереву.

В связи с тем, что VT оперирует узлами как связанным списком (каждый узел сам по себе можно представить связанным списком), написание разнообразных циклов для прохода по дереву может занять некоторое время, превратившись в утомительную рутину. К счастью, в VT предусмотрен ряд очень удобных и полезных функций для прохода по дереву, которые превращают эту задачу в пару нажатий клавиш. Их около 30 (!) и уж с ними Вы точно не заблудитесь.

Ниже приведён список всех этих функций с описанием входных параметров (если есть), принципом работы и её результатом.

- **GetFirst**, **GetLast (Node: PVirtualNode = nil)** – возвращают самый первый или самый последний узел дерева соответственно. Самый первый узел - узел RootNode.FirstChild. Это корневой узел с нулевым индексом по отношению к RootNode. Самый последний узел – самый последний дочерний узел у самого последнего корневого узла. Функция GetLast позволяет указать узел в качестве входного параметра. Если таковой будет указан, то функция вернёт самый последний узел по отношению к указанному.
- **GetFirstChecked (State: TCheckState)** – возвращает первый попавшийся узел, состояние отметки которого эквивалентно указанному во входном параметре.
- **GetFirstChild (Node: PVirtualNode), GetLastChild (Node: PVirtualNode)** – возвращают первый и последний дочерний узел родительского узла, указанного во входном параметре соответственно.
- **GetFirstCutCopy** – возвращает первый попавшийся узел, определённый для копирования или вырезания в буфер обмена.
- **GetFirstInitialized, GetLastInitialized (Node: PVirtualNode = nil)** – первая функция возвращает первый попавшийся узел, который содержит флаг **vsInitialized**, т.е. который был инициализирован. Вторая функция возвращает самый последний узел с **vsInitialized** флагом по отншению к родительскому узлу, указанному во входном параметре.
- **GetFirstNoInit, GetLastNoInit (Node: PVirtualNode = nil)** – то же самое, что и предыдущии функции, только делают всё с точностью до наоборот.
- **GetFirstSelected** – возвращает первый попавшийся узел, который содержит флаг **vsSelected**, т.е. который является выделенным на данный момент.
- **GetFirstVisible, GetLastVisible (Node: PVirtualNode = nil)** – возвращают первый попавшийся и последний узел, который содержит флаг **vsVisible** соответственно, т.е. который виден на данный момент. Вторая функция позволяет указать родительскийузел во входном параметре относительно которого будет произведён поиск узла.
- **GetFirstVisibleNoInit, GetLastVisibleNoInit (Node: PVirtualNode = nil)** – то же самое, что и предыдущии функции, но поиск производится только в узлах без флага **vsInitialized**.
- **GetFirstVisibleChild (Node: PVirtualNode), GetLastVisibleChild (Node: PVirtualNode)** – возвращают первый попавшийся и последний дочерний узел родительского узла, указанного во входном параметре, который выделен на данный момент соответственно.
- **GetFirstVisibleChildNoInit (Node: PVirtualNode), GetLastVisibleChildNoInit (Node: PVirtualNode)** – то же самое, что и предыдущии функции, но поиск производится только в узлах без флага **vsInitialized**.
- **GetNext (Node: PVirtualNode), GetPrevious (Node: PVirtualNode)** – возвращают следующий и предыдущий узлы, смежные (соседние) с узлом, указанном во входном параметре. Эти функции отличаются от почти аналогичных GetNextSibling и GetPreviousSibling. В случае если по отношению к указанному узлу больше нет соседей, то GetNext вернёт следующий узел, смежный с родительским узлом, а GetPrevious вернёт сам родительский узел по отношению к указанному. Также если у указанного узла есть дочерние узлы, то GetNext вернёт первый дочерний узел, а GetPrevious самый последний дочерний узел предыдущего узла. На самом деле, со слов это довольно трудно понять, просто попробуйте поприменять эти функции на практитке.
- **GetNextChecked (Node: PvirtualNode; State: TCheckState = csCheckedNormal)** – возвращает следующий узел по отношению к указанному с отметкой, эквивалентной указанной во входном параметре.
- **GetNextCutCopy (Node: PVirtualNode)** – возвращает следующий узел по отношению к указанному, который определён пользователем для вырезания или копирования. Т.е. узел, содержащий флаг **vsCutOrCopy**.
- **GetNextInitialized (Node: PVirtualNode), GetPreviousInitialized (Node: PVirtualNode)** – возвращают следующий и предыдущий инициализированные узлы по отношению к указанному во входном параметре соответственно.

- **GetNextNoInit (Node: PVirtualNode), GetPreviousNoInit (Node: PVirtualNode)** - тоже самое, что и GetNext. Отличие в том, что при GetNext узлы инициализируются, а при GetNextNoInit - нет (спасибо forever за эту информацию).
- **GetNextSelected (Node: PVirtualNode)** - возвращает следующий выделенный узел по отношению к указанному во входном параметре.
- **GetNextSibling (Node: PVirtualNode), GetPreviousSibling (Node: PVirtualNode)** - возвращают следующий соседний узел по отношению к указанному во входном параметре. Эти функции отличаются от GetNext и GetPrevious тем, что никаких перескоков через родительские узлы не осуществляется. Диапазон действия функций ограничевается только дочерними узлами. Если соседних узлов по отношению к указанному больше не будет, то функции вернут **nil**.
- **GetNextVisible (Node: PVirtualNode), GetPreviousVisible (Node: PVirtualNode)** - возвращают следующий видимый узел по отношению к указанному во входном параметре. Т.е. узел, содержащий флаг **vsVisible**.
- **GetNextVisibleNoInit (Node: PVirtualNode), GetPreviousVisibleNoInit (Node: PVirtualNode)** - то же самое, но поиск производится только в узлах без **vsInitialized** флага.
- **GetNextVisibleSibling (Node: PVirtualNode), GetPreviousVisibleSibling (Node: PVirtualNode)** - то же самое, что и GetNextVisible и GetPreviousSibling, но действие функций ограничено только соседними узлами по отношению к указанному.
- **GetNextVisibleSiblingNoInit (Node: PVirtualNode), GetPreviousVisibleSiblingNoInit (Node: PVirtualNode)** - то же самое, но поиск производится только в узлах без **vsInitialized** флага.
- **GetSortedCutCopySet (Resolve: Boolean)** - возвращает массив узлов, определённых для вырезания или копирования в буфер обмена. Елси входной параметр равен True, то в массив не будут включены дочерние узлы вырезаемых или копируемых узлов. Узлы располагаются в массиве в порядке их появления в дереве.
- **GetSortedSelection (Resolve: Boolean)** - возвращает массив выделенных узлов. Очень важная функция при реализации Drag&Drop операций с деревом.

Стоит упомянуть об ещё одной очень важной функции для прохода по дереву - **IterateSubtree**.

```
function TBaseVirtualTree.IterateSubtree(Node: PVirtualNode; Callback:
TVTGetNodeProc; Data: Pointer; Filter: TVirtualNodeStates = []; DoInit:
Boolean = False; ChildNodesOnly: Boolean = False): PVirtualNode;
```

Эта функция без особых забот позволяет пройтись по всем узлам дерева и для каждого выполнить какое-то действие, используя callback-процедуру. Кроме того, с помощью фильтра можно указать для каких именно узлов нужно вызывать callback-процедуру. **Node** задаёт начальный узел, с которого начнётся проход по дереву. **DoInit** позволяет указать нужно ли проинициализировать узел перед тем, как вызывать callback-процедуру для него. **ChildNodesOnly** позволяет указать следует ли вызывать callback-процедуру для всех узлов или только для дочерних. Функция возвращает последний узел, обработанный callback-процедурой.

Следующий пример демонстрирует подсчёт выделенных узлов в дереве с помощью **IterateSubtree**.

```
procedure TForm1.IterateNodes(Sender: TBaseVirtualTree; Node: PVirtualNode;
  Data: Pointer; var Abort: Boolean);
begin
  Inc(PInteger(Data)^);
end;

procedure TForm1.VTChange(Sender: TBaseVirtualTree; Node: PVirtualNode);
var
  Sel: Integer;
begin
  Sel := 0;
  VT.IterateSubtree(nil, IterateNodes, @Sel, [vsSelected], False, False);
  Lbl.Caption := 'Выбрано: ' + IntToStr(Sel);
end;
```

Как видите, callback-процедура также позволяет указать указатель на какую-нибудь переменную. Кроме того, если установить **Abort** в True, то процесс прохода по дереву прервётся. Используйте этот параметр для выхода из цикла поиска.

Проект Figure 1.2, демонстрирующий работу с наиболее часто используемыми событиями VT находится в папке Fig 1.2.

В этой сравнительно большой главе мы разобрались с наиболее часто используемыми событиями и методами VT. Но это ещё далеко не конец. Далее мы ещё больше углубимся в недра VT и реализуем более сложные вещи, такие как Drag&Drop, взаимодействие с шеллом, работа с буфером обмена и сохранение и загрузка дерева из внешних источников.

3.0. Принципы реализации Drag&Drop, взаимодействие с шеллом, работа с буфером обмена.

Бесспорно, на лавры одной из самых удобных возможностей интерфейса, которая когда-либо была изобретена, имеет все основания претендовать Drag&Drop (за исключением, пожалуй, двойного клика :)), позволяющая выполнить многие операции без клавиатуры простыми манипуляциями мышкой. И бесспорно, она же и самая нетривиальная в реализации задача. VT в этом плане не исключение. Несмотря на то, что в компоненте максимально возможно реализована поддержка Drag&Drop различными событиями, свойствами и интерфейсами, нам придётся делать основную часть работы самим.

3.1. Подходы к реализации Drag&Drop.

VT позволяет реализовать Drag&Drop двумя базовыми подходами (свойство **DragType**):

- **dtOLE** – основан на системных методах и функциях, работает через ActiveX. Настоятельно рекомендуется к использованию, т.к. полностью поддерживается Windows и позволяет реализовать больше возможностей с минимумом проблем. Работает через Windows-интерфейсы IDataObject, IDragManager и т.д.
- **dtVCL** – подход, навязанный Borland специально ради облегчения проектирования кроссплатформенных приложений. Про него можно точно сказать: одно лечит, а другое калечит. Главный его недостаток в том, что получатель должен иметь как можно более полные сведения об отправителе и его данных. Таким образом, в нём отсутствует поддержка Drag&Drop между приложениями. Всё будет происходить только в контексте Вашей программы. В ряде случаев это заставляет писать прорву кода только для того, чтобы отделить один класс от другого. Я уже и не говорю о возможностях их кардинального различия. Старайтесь избегать VCL подход. Он теоретически и не является Drag&Drop вовсе.

Думаю, следует побольше сказать о выборе из этих двух методов.

OLE позволяет получать данные из любых приложений, в том числе и из своего. Причём VT при использовании OLE способен также получать данные и из VCL контролов Delphi, обученных только VCL Drag&Drop от Borland.

VCL позволяет VT общаться только с контролами, которые используют VCL подход. Но это, тем не менее, не значит, что VT не будет способен работать с OLE данными, если ему выставить dtVCL свойство.

Тогда зачем этот параметр **DragType** вообще, спросите Вы, если он всё равно ничего не меняет? Сейчас всё станет ясно.

3.2. Опция toAcceptOLEDrop.

Данный сеттинг находится в TreeOptions.MiscOptions. Он как раз и позволяет разрешить или запретить вставку OLE данных. Если опция отключена, то дерево с **dtOLE** параметром вообще перестанет воспринимать Drag&Drop, тогда как **dtVCL**-дерево будет ещё способно к VCL Drag&Drop в Вашем приложении. Если же опция включена, то дерево будет способно работать с любым типом Drag&Drop. Всё это станет Вам ещё более понятно, когда мы напишем пример, который, кстати, будет поддерживать оба этих подхода.

3.3. Параметры Drag&Drop.

Помимо **DragType**, VT позволяет указать различные параметры операции. Рассмотрим их:

- **ClipboardFormats** – содержит список форматов, распознаваемых VT для **OLE** метода (В VCL всё придётся делать ручками). Как минимум, должен быть включён Plain Text и Virtual Tree Data.
- **DefaultPasteMode** – положение вставляемой ветки по умолчанию.
- **DragWidth, DragHeight** – размеры прозрачной картинки, отображающей краткое содержимое перетаскиваемых объектов. Чем больше размер, тем больше потребуется мощности процессора для обсчёта прозрачности.
- **DragImageKind** – позволяет указать будут ли в прозрачной картинке отображаться картинки узлов VT.
- **DragMode** – перетаскивание начнётся автоматически при dmAutomatic или потребует ручного управления при dmManual. Только не подумайте, что dmAutomatic всё сделает за Вас :). Нет, этот режим часто применяется вместе с методом OLE, тогда как dmManual предназначен для VCL метода.
- **DragOperations** – перечень операций для работы с Drag&Drop.

3.4. События Drag&Drop.

Вот они:

- **OnCreateDataObject** - вызывается, когда drag manager'у необходим экземпляр класса TDataObject с заполненными полями.
- **OnCreateDragManager** - позволяет пользователю создавать свои собственные drag manager'ы для дерева. К сожалению, Я не имею опыта работы с этим сообщением.
- **OnGetUserClipboardFormats** - позволяет заполнить массив форматами данных, которые Вы способны обрабатывать. Используется совместно с OnRenderOLEData.
- **OnRenderOLEData** - используется для обработки данных, которые не поддерживаются встроенным обработчиком VT.
- **OnDragAllowed** - позволяет разрешить или запретить перетаскивание элементов.
- **OnDragDrop** - основное событие, происходит непосредственно после того, как была отпущена кнопка мыши и были получены данные.
- **OnDragOver** - происходит при наведении указателя на элемент дерева. Позволяет разрешить или запретить перетаскивание в этот элемент.
- **OnEndDrag** - Событие, следующее после OnDragDrop. Оповещает об окончательном завершении операции.
- **OnStartDrag** - оповещает о начале перетаскивания.
- **OnNodeCopied** - событие, связанное с CopyToClipboard/CutToClipboard методами дерева. Происходит сразу после копирования узла.
- **OnNodeCopying** - аналогично предыдущему, но происходит до вышеупомянутых методов и позволяет также запретить копирование.

3.5. Пишем Drag&Drop приложение с VT.

Мы подошли к решающему шагу - написанию своего первого Drag&Drop приложения с VT. Я решил в этом примере одновременно показать реализацию не только Drag&Drop, но и вставки из буфера обмена и работы с шеллом одновременно. Почему Я смешал всё в одну кашу? Дело в том, что в VT имеется собственный механизм для копирования и вставки своих узлов. Но что если в буфере обмена не узлы VT, а какие-то другие данные? Тогда вставку можно будет организовать через Drag&Drop интерфейсы, что мы и сделаем.

Нам понадобятся:

- Дерево VT с dtOLE и dmAutomatic свойствами.
- Дерево VT с VCL и dmManual свойствами.
- TRichEdit в качестве источника OLE Drag&Drop данных.
- TListBox в качестве источника VCL Drag&Drop данных. Не забудьте также выставить ему dmAutomatic свойство.
- У обоих VT-деревьев установите все ClipboardFormats в True.
- Желательно также включить **toMultiSelect** в TreeOptions.SelectionOptions, чтобы можно было выделять несколько узлов разом.
- 3 действия (**TAction**) для копирования, вырезания и вставки. Не забудьте им выставить соответствующие Shortcut, чтобы можно было вызывать их прямо с клавиатуры без всяких меню и кнопок.

Данные узла возьмём самые маленькие, чтобы не отвлекаться на лишнее:

```
type
  PItemNode = ^TItemNode;

  TItemNode = record
    Name: WideString;
  end;
```

Заполняем контролы:

```
procedure TfrmMain.FormCreate(Sender: TObject);
var
  i: Integer;
begin
  VT.NodeDataSize := SizeOf(TItemNode);
  VT2.NodeDataSize := SizeOf(TItemNode);
  VT.RootNodeCount := 30;
  VT2.RootNodeCount := 30;
  RichEdit.Lines.LoadFromFile('rtf.rtf');
  for i := 0 to 9 do
    ListBox.Items.Add(Format('String %d', [i]));
end;
```

Заметили, что мы в вышеприведённом коде не заполнили узлы никакими данными? До этого мы в цикле обрабатывали каждый узел. На этот раз Я решил показать пример работы с ещё одним полезным событием VT - **OnInitNode**. Оно позволяет заполнить узел данными, если Вы не сделали этого при создании. Помимо этого, Вам будет также предложено задать различные свойства для узла. В нашем случае обработчик этого события для обоих деревьев будет выглядить так:

```
procedure TfrmMain.VTInitNode(Sender: TBaseVirtualTree; ParentNode,
  Node: PVirtualNode; var InitialStates: TVirtualNodeInitStates);
var
  ItemNode: PItemNode;
begin
  ItemNode := Sender.GetNodeData(Node);
  if Assigned(ItemNode) then
    if Length(ItemNode^.Name) = 0 then
      ItemNode^.Name := 'Node Index № ' + IntToStr(Node.Index);
end;
```

Как видите, в нашем случае гораздо удобнее использовать именно это событие, нежели делать всё непосредственно при создании узлов. Ведь все нам необходимые данные (индекс узла) мы получаем, не отходя от кассы, прямо на месте. Данное событие не является обязательным. Впредь Вы вольны будете решать, какой подход Вам будет удобнее.

Следующие два обработчика Вам уже знакомы, назначьте их обоим деревьям:

```
procedure TfrmMain.VTGetText(Sender: TBaseVirtualTree; Node: PVirtualNode;
  Column: TColumnIndex; TextType: TVSTTextType; var CellText: WideString);
var
  ItemNode: PItemNode;
begin
  ItemNode := Sender.GetNodeData(Node);
  if Assigned(ItemNode) then
    CellText := ItemNode^.Name;
end;
```

```
procedure TfrmMain.VTNewText(Sender: TBaseVirtualTree; Node: PVirtualNode;
  Column: TColumnIndex; NewText: WideString);
var
  ItemNode: PItemNode;
begin
  ItemNode := Sender.GetNodeData(Node);
  if Assigned(ItemNode) then
    ItemNode^.Name := NewText;
end;
```

Вот так выглядят обработчики копирования и вырезания для действий из **TActionList**:

```
procedure TfrmMain.BtnCopyClick(Sender: TObject);
begin
  if ActiveControl = VT then
    VT.CopyToClipBoard
  else if ActiveControl = VT2 then
    VT2.CopyToClipBoard
  else if ActiveControl = RichEdit then
    RichEdit.CopyToClipboard;
end;
```

```
procedure TfrmMain.BtnCutClick(Sender: TObject);
begin
  if ActiveControl = VT then
    VT.CutToClipBoard
  else if ActiveControl = VT2 then
    VT2.CutToClipBoard
  else if ActiveControl = RichEdit then
    RichEdit.CutToClipBoard;
end;
```

А вот процедура вставки уже будет различаться. Помните, Я говорил, что вставку OLE данных мы сделаем через те же интерфейсы, что и Drag&Drop? Вот что у нас получится:

```
procedure TfrmMain.BtnPasteClick(Sender: TObject);
var
  DataObject: IDataObject;
  EnumFormat: IEnumFormatEtc;
  Format: TFormatEtc;
  Formats: TFormatArray;
  Fetched: Integer;
  Tree: TVirtualStringTree;
begin
  if ActiveControl is TVirtualStringTree then
  begin
    Tree := ActiveControl as TVirtualStringTree;
    // Попробуем сначала вставить данные простым методом VT.
    if not Tree.PasteFromClipboard then
    begin
      // Если VT сам не смог справиться со вставкой, значит пришли данные
      // несколько другого типа, чем просто TVirtualNode узлы.
      // Это может быть, к примеру, текст или картинка.
      // Сейчас мы это узнаем, покопавшись в форматах буфера обмена.
      OLEGetClipboard(DataObject);
      // Получить список доступных в буфере обмена форматов в массив Formats.
      if Succeeded(DataObject.EnumFormatEtc(DATADIR_GET, EnumFormat)) then
      begin
        EnumFormat.Reset;
        while EnumFormat.Next(1, Format, @Fetched) = S_OK do
        begin
          SetLength(Formats, Length(Formats) + 1);
          Formats[High(Formats)] := Format.cfFormat;
        end;
        InsertData(Tree, DataObject, Formats, DROPEFFECT_COPY,
Tree.DefaultPasteMode);
      end;
    end;
  end
  else if ActiveControl is TRichEdit then
    RichEdit.PasteFromClipboard;
```

```
end;
```

Теперь следуют собственно обработчики Drag&Drop. Все комментарии Я поместил в код. Для обоих деревьев два одинаковых обработчика. **onDragDrop** и **onDragOver**:

```
//----------------------------------------------------------------------------
// На нас что-то кинули…
//----------------------------------------------------------------------------
procedure TfrmMain.VTDragDrop(Sender: TBaseVirtualTree; Source: TObject;
  DataObject: IDataObject; Formats: TFormatArray; Shift: TShiftState;
  Pt: TPoint; var Effect: Integer; Mode: TDropMode);

  // Определяем как поступать с данными. Перемещать, копировать или ссылаться
  procedure DetermineEffect;
  begin
    // Нажаты ли какие-нибудь управляющие клавиши?
    if Shift = [] then
    begin
      // Неа, не нажаты
      // Тогда, если отправитель и получатель - одинаковые объекты (например,
      // если узлы перемещаются из одного и того же дерева), то
      // надо переместить узлы, в противном случае - копировать.
      if Source = Sender then
        Effect := DROPEFFECT_MOVE
      else
        Effect := DROPEFFECT_COPY;
    end
    else begin
      // Нажаты. В зависмости от комбинации решаем что делать
      if (Shift = [ssAlt]) or (Shift = [ssCtrl, ssAlt]) then
        Effect := DROPEFFECT_LINK
      else
        if Shift = [ssCtrl] then
          Effect := DROPEFFECT_COPY
        else
          Effect := DROPEFFECT_MOVE;
    end;
  end;

var
  Attachmode: TVTNodeAttachMode;
  Nodes: TNodeArray;
  i: Integer;
begin
  Nodes := nil;
  // Определяем, куда добавлять узел в зависимости от того, куда была
  // брошена ветка.
  case Mode of
    dmAbove:
      AttachMode := amInsertBefore;
    dmOnNode:
      AttachMode := amAddChildLast;
    dmBelow:
      AttachMode := amInsertAfter;
  else
    AttachMode := amNowhere;
  end;
  if DataObject = nil then
  begin
    // Если не пришло интерфейса, то вставка проходит через VCL метод
```

```
    if Source is TVirtualStringTree then
    begin
      // Вставка из VT. Можем спокойно пользоваться его методами
      // копирования и перемещения.
      DetermineEffect;
      // Получаем список узлов, которые будут участвовать в Drag&Drop
      Nodes := VT2.GetSortedSelection(True);
      // И работаем с каждым
      if Effect = DROPEFFECT_COPY then
      begin
        for i := 0 to High(Nodes) do
          VT2.CopyTo(Nodes[i], Sender.DropTargetNode, AttachMode, False);
      end
      else
        for i := 0 to High(Nodes) do
          VT2.MoveTo(Nodes[i], Sender.DropTargetNode, AttachMode, False);
    end
    else if Source is TListBox then
    begin
      // Вставка из объекта какого-то другого класса
      AddVCLText(Sender as TVirtualStringTree,
        (Source as TListBox).Items.Strings[(Source as TListBox).ItemIndex],
        AttachMode);
    end;
  end
  else begin
    // OLE drag&drop.
    // Effect нужен для передачи его источнику drag&drop, чтобы тот решил
    // что он будет делать со своими перетаскиваемыми данными.
    // Например, при DROPEFFECT_MOVE (перемещение) их нужно будет удалить,
    // при копировании - сохранить.
    if Source is TBaseVirtualTree then
      DetermineEffect
    else begin
      if Boolean(Effect and DROPEFFECT_COPY) then
        Effect := DROPEFFECT_COPY
      else
        Effect := DROPEFFECT_MOVE;
    end;
    InsertData(Sender as TVirtualStringTree, DataObject, Formats, Effect,
AttachMode);
  end;
end;

//---------------------------------------------------------------------------
// В этом событии мы должны проверить есть ли среди перетаскиваемых веток
// родитель ветки, в которую происходит перетаскивание. Ведь нельзя
// же ветку-родитель перетащить в её дочерние элементы :)
//---------------------------------------------------------------------------
procedure TfrmMain.VTDragOver(Sender: TBaseVirtualTree; Source: TObject;
  Shift: TShiftState; State: TDragState; Pt: TPoint; Mode: TDropMode;
  var Effect: Integer; var Accept: Boolean);

  // Возвращает True, если AParent - дочерний узел ANode.
  function IsNodeParent(AParent, ANode: PVirtualNode): Boolean;
  var
    NextParent: PVirtualNode;
  begin
    NextParent := AParent;
    repeat
      NextParent := NextParent.Parent;
    until
```

```
      (NextParent = Sender.RootNode) or (NextParent = nil) or
        (NextParent = ANode);
    Result := ANode = NextParent;
  end;

var
  i: Integer;
  Nodes: TNodeArray;
begin
  Accept := True;
  if (Assigned(Sender.DropTargetNode)) and
    (Sender.DropTargetNode <> Sender.RootNode) then
    Nodes := (Sender as TVirtualStringTree).GetSortedSelection(True);
  if Length(Nodes) > 0 then
  begin
    for i := 0 to Length(Nodes) - 1 do
    begin
      Accept :=
        // Узел не должен быть родителем ветки, в которую производится
        // вставка
        (not IsNodeParent(Sender.DropTargetNode, Nodes[i]))
        // Также, узел не должен равняться ветке-местоназначению вставки.
        // Т.е. мы должны запретить вставку узла в самого себя.
        and (not(Sender.DropTargetNode = Nodes[i]));
      // Отключаем вставку, если хотя бы одно из условий вернуло False
      if not Accept then
        Exit;
    end;
  end;
end;
```

Этот обработчик **onDragAllowed** назначьте только дереву с VCL Drag&Drop:

```
procedure TfrmMain.VTDragAllowed(Sender: TBaseVirtualTree; Node:
PVirtualNode;
  Column: TColumnIndex; var Allowed: Boolean);
begin
  Allowed := True;
end;
```

Метод **InsertData** предназначен непосредственно для вставки новых OLE данных в дерево после того, как мы их получили. Он будет искать среди полученных форматов тот, который способен обработать. В нашем случае это ЮНИКОД-текст и файлы, брошенные на нас из Explorer'а. Вот как он выглядит:

```
procedure TfrmMain.InsertData(Sender: TVirtualStringTree;
  DataObject: IDataObject; Formats: TFormatArray;
  Effect: Integer; Mode: TVTNodeAttachMode);
var
  FormatAccepted: Boolean; // True, если принятые данные уже обработались
  i: Integer;
begin
  // Ищем в переданных форматах тот, который можем обработать
  FormatAccepted := False;
  for i := 0 to High(Formats) do
  begin
    case Formats[i] of
      CF_HDROP: // Прибыл список файлов из Explorer'а.
      begin
```

```
        if not FormatAccepted then
        begin
          AddFile(DataObject, Sender as TVirtualStringTree, Mode);
          FormatAccepted := True;
        end;
      end;
      CF_UNICODETEXT: // ЮНИКОД-текст
      begin
        if not FormatAccepted then
        begin
          AddUnicodeText(DataObject, Sender as TVirtualStringTree, Mode);
          FormatAccepted := True;
        end;
      end;
      else if Formats[i] = CF_VIRTUALTREE then
      // Родной формат VT. Обрабатывает вставку своих же
      // TVirtualNode-узлов.
      begin
        if not FormatAccepted then
        begin
          Sender.ProcessDrop(DataObject, Sender.DropTargetNode, Effect,
Mode);
          FormatAccepted := True;
        end;
      end;
    end;
  end;
end;
```

Теперь посмотрим, как выглядят процедуры **AddVCLText**, **AddFile** и **AddUnicodeText**. Все они предназначены для обработки какого-то одного типа пришедших данных. Вот обработчики каждого из них с подробным описанием каждого действия:

```
//---------------------------------------------------------------------------
// Метод предназначен для обработки ЮНИКОД-текста, хранящегося в буфере
// обмена.
// Хранимый текст получается и разделяется на линии, которые затем
// вставляются в дерево в качестве новых узлов.
//---------------------------------------------------------------------------
procedure TfrmMain.AddUnicodeText(DataObject: IDataObject;
  Target: TVirtualStringTree; Mode: TVTNodeAttachMode);
var
  // Структура, которая будет запрашивать указатель на глобальные данные
  FormatEtc: TFormatEtc;
  // Структура с данными
  Medium: TStgMedium;
  // Указатель на строку
  OLEData,
  // Для прохода цикла с целью разделения текста на строки
  Head, Tail: PWideChar;
  // Узел, получающий новую ветку
  TargetNode,
  // Новая ветка
  Node: PVirtualNode;
  // Её данные
  Data: PItemNode;
begin
  if Mode <> amNowhere then
  begin
    // Заполняем струкутру для получения ЮНИКОД-текста
```

```
    with FormatEtc do
    begin
      cfFormat := CF_UNICODETEXT;
      // Нет у нас никакого девайса...
      ptd := nil;
      // Нам нужно содержание (текст в данном случае)
      dwAspect := DVASPECT_CONTENT;
      // Нет страницы для мультистраничных данных
      lindex := -1;
      // Мы будем получать указатель на данные через глобальную память
      tymed := TYMED_HGLOBAL;
    end;
    // Есть ли ЮНИКОД-текст для нашего запроса?
    if DataObject.QueryGetData(FormatEtc) = S_OK then
    begin
      // Опа, есть, можем получить данные
      if DataObject.GetData(FormatEtc, Medium) = S_OK then
      begin
        // Вот они:
        OLEData := GlobalLock(Medium.hGlobal);
        if Assigned(OLEData) then
        begin
          Target.BeginUpdate;
          // Выбираем место для вставки, если переданное = nil
          TargetNode := Target.DropTargetNode;
          if TargetNode = nil then
            TargetNode := Target.FocusedNode;
          // Разбиваем текст на строки
          Head := OLEData;
          try
            while Head^ <> #0 do
            begin
              Tail := Head;
              while not (Tail^ in [WideChar(#0), WideChar(#13),
WideChar(#10), WideChar(#9)]) do
                Inc(Tail);
              if Head <> Tail then
              begin
                // Добавляем новую ноду, если есть хотя бы один символ
                // для строки.
                Node := Target.InsertNode(TargetNode, Mode);
                Data := Target.GetNodeData(Node);
                Data^.Name := Head;
                SetLength(Data^.Name, (Tail - Head));
              end;
              // Пропускаем табы
              if Tail^ = #9 then
                Inc(Tail);
              // Символы переноса каретки и конца строки
              if Tail^ = #13 then
                Inc(Tail);
              if Tail^ = #10 then
                Inc(Tail);
              // Шагаем дальше
              Head := Tail;
            end;
          finally
            GlobalUnlock(Medium.hGlobal);
            Target.EndUpdate;
          end;
        end;
        // Вот это лучше не забывать делать
```

```
        ReleaseStgMedium(Medium);
      end;
    end;
  end;
end;
```

```
//---------------------------------------------------------------------------
// Получение имён файлов, брошеных на нас из Explorer'а.
//---------------------------------------------------------------------------
procedure TfrmMain.AddFile(DataObject: IDataObject;
  Target: TVirtualStringTree; Mode: TVTNodeAttachMode);
var
  FormatEtc: TFormatEtc;
  Medium: TStgMedium;
  OLEData: PDropFiles;
  Files: PChar;
  Str: String;
  TargetNode,
  Node: PVirtualNode;
  Data: PItemNode;
begin
  if Mode <> amNowhere then
  begin
    // На этот раз нас интересует есть ли в буфере CF_HDROP формат
    with FormatEtc do
    begin
      cfFormat := CF_HDROP;
      ptd := nil;
      dwAspect := DVASPECT_CONTENT;
      lindex := -1;
      tymed := TYMED_HGLOBAL;
    end;
    if DataObject.QueryGetData(FormatEtc) = S_OK then
    begin
      if DataObject.GetData(FormatEtc, Medium) = S_OK then
      begin
        OLEData := GlobalLock(Medium.hGlobal);
        if Assigned(OLEData) then
        begin
          Target.BeginUpdate;
          TargetNode := Target.DropTargetNode;
          if TargetNode = nil then
            TargetNode := Target.FocusedNode;
          try
            // А вот с этим Я долго мучался. В Microsoft зажрались и не дали
            // примера работы с DROPFILES структурой.
            // Оказывается, список брошенных файлов хранится в адресе
            // структуры
            // + offset, который и есть OLEData^.pFiles.
            Files := PChar(OLEData) + OLEData^.pFiles;
            // Список оканчивается двойным null символом
            while Files^ <> #0 do
            begin
              if OLEData^.fWide then
              begin
                Str := PWideChar(Files);
                // +1 нужен для того, чтобы перешагнуть null символ одного
                // из имён файлов в списке
                Inc(Files, (Length(PWideChar(Files)) + 1)*SizeOf(WideChar));
              end
              else begin
```

```
                Str := Files;
                // Аналогично
                Inc(Files, (Length(PChar(Files)) + 1)*SizeOf(Char));
              end;
              Node := Target.InsertNode(TargetNode, Mode);
              Data := Target.GetNodeData(Node);
              Data^.Name := Str;
            end;
          finally
            GlobalUnlock(Medium.hGlobal);
            Target.EndUpdate;
          end;
        end;
        ReleaseStgMedium(Medium);
      end;
    end;
  end;
end;
```

```
//---------------------------------------------------------------------------
// Метод для получения строки из VCL контролов.
// Режем на линии.
//---------------------------------------------------------------------------
procedure TfrmMain.AddVCLText(Target: TVirtualStringTree;
  const Text: WideString; Mode: TVTNodeAttachMode);
var
  Head, Tail: PWideChar;
  TargetNode,
  Node: PVirtualNode;
  Data: PItemNode;
begin
  if Mode <> amNoWhere then
  begin
    Target.BeginUpdate;
    try
      TargetNode := Target.DropTargetNode;
      if TargetNode = nil then
        TargetNode := Target.FocusedNode;
      Head := PWideChar(Text);
      while Head^ <> #0 do
      begin
        Tail := Head;
        while not (Tail^ in [WideChar(#0), WideChar(#13), WideChar(#10),
WideChar(#9)]) do
          Inc(Tail);
        if Head <> Tail then
        begin
          Node := Target.InsertNode(TargetNode, Mode);
          Data := Target.GetNodeData(Node);
          SetString(Data^.Name, Head, Tail - Head);
        end;
        if Tail^ = #9 then
          Inc(Tail);
        if Tail^ = #13 then
          Inc(Tail);
        if Tail^ = #10 then
          Inc(Tail);
        Head := Tail;
      end;
    finally
      Target.EndUpdate;
```

```
    end;
  end;
end;
```

Всё! Мы сделали это. Можете теперь скомпилировать пример и проверить его работоспособность. В качестве теста, попробуйте копировать и вырезать узлы, перемещать их между деревьями, перетаскивать текст из TRichEdit, TListBox, MSWord, Visual Studio, Total Commander, выделите несколько папок и файлов в Explorer и киньте их на деревья. Можете попробовать взаимодействовать и с другими приложениями, а также перетаскивать узлы из деревьев в TRichEdit или другие приложения. Всё в итоге должно работать на ура. Почти всегда реализованных нами методов вполне хватает, разве что обработку вставки графики мы не реализовали, но это можете оставить уже себе в качестве тренировки :). Поздравляю, Ваше приложение полностью совместимо Drag&Drop!

Проект Figure 1.3, демонстрирующий работу с Drag&Drop находится в папке Fig 1.3.

4.0. Создание собственных редакторов данных. Работа с интерфейсом IVTEditLink.

VT предоставляет замечательную возможность создавать свои собственные редакторы данных, кроме обычного текстового поля для редактирования текста. Эту возможность можно использовать, чтобы облегчить пользователю работу с какими-то специализированными данными. Например, при изменении цвета можно использовать ComboBox со списком цветов, а при изменении даты - вызывать календарь. Конечно, это непременно прибавит профессиональности вашей программе.

Чтобы как-то обобщить редактирование различных типов данных и позволить взаимодействовать редакторам дерева с приложением, компонент использует интерфейс **IVTEditLink**. Давайте создадим небольшой пример, где постараемся реализовать собственные редакторы для различных типов данных.

4.1. Пишем класс, реализующий интерфейс IVTEditLink.

Для начала определимся, какие форматы данных, а следственно редакторы, мы будем использовать в примере. Думаю, этого будет достаточно для начала:

```
type
  TVTEditorKind = (
    ekString, // TEdit
    ekMemo, // TMemo
    ekComboBox, // TComboBox
    ekColor, // TColorBox
    ekDate, // TDateTimePicker
    ekMask, // TMaskedEdit
    ekProgress // TProgressBar
  );
```

Мы напишем один класс для реализации интерфейса для каждого типа редактора. Это наиболее рациональный подход вместо того, чтобы для каждого редактора создавать свой класс, реализующий один и тот же интерфейс. Вот как он будет выглядеть:

```
TVTCustomEditor = class(TInterfacedObject, IVTEditLink)
private
  FEdit: TWinControl;         // Базовый класс для каждого типа редактора
  FTree: TVirtualStringTree; // Ссылка на дерево, вызвавшее редактирование
  FNode: PVirtualNode;        // Редактируемый узел
  FColumn: Integer;           // Его колонка, в которой оно происходит
protected
  procedure EditKeyDown(Sender: TObject; var Key: Word; Shift: TShiftState);
public
  destructor Destroy; override;

  function BeginEdit: Boolean; stdcall;
  function CancelEdit: Boolean; stdcall;
  function EndEdit: Boolean; stdcall;
  function GetBounds: TRect; stdcall;
  function PrepareEdit(Tree: TBaseVirtualTree; Node: PVirtualNode;
    Column: TColumnIndex): Boolean; stdcall;
  procedure ProcessMessage(var Message: TMessage); stdcall;
  procedure SetBounds(R: TRect); stdcall;
end;
```

Как видите, все специализированные редакторы будут потомками TWinControl. С помощью типа TVTEditorKind мы будем определять, какой конкретно класс редактора будет создаваться. В комментариях, напротив полей перечисления, указан класс редактора для каждого поля.

Теперь нам необходимо решить, как будет выглядеть структура данных для каждого узла. Нам необходимо где-то хранить строковую переменную, которая в соответствии с типом редактора будет трансформироваться в нужный тип данных, а также переменную типа TVTEditorKind, чтобы мы могли знать какой тип редактора будет создан для данного конкретного узла.

Получаем:

```
type
  PVTEditNode = ^TVTEditNode;

  TVTEditNode = record
    Kind: TVTEditorKind;
    Value: String;
    Changed: Boolean;
  end;
```

Сюда Я также добавил поле Changed, чтобы определить, что узел был отредактирован и не содержит больше своего значения по умолчанию. Это необязательный параметр.

Наш пример будет содержать 2 колонки (параметр-значение) и 7 узлов для каждого типа редактора. Чтобы не отвлекаться и не писать лишнего кода, определим для данных 3 массива: типы редакторов для каждого из семи узлов, значения по умолчанию для поля Changed и текст для первой колонки.

```
  ValueTypes: array[0..6] of TVTEditorKind = (
    ekString, // Имя
    ekMemo, // Описание
    ekComboBox, // Тип
    ekColor, // Цвет
    ekDate, // Дата
    ekMask, // Маска
    ekProgress // Процесс
  );
  // Значения по умолчанию
  DefaultValues: array[0..6] of String = (
    'Свитер',
    'Мягкий и тёплый.',
    'Шерсть',
    'clRed',
    '24.06.2006',
    '798-77-66',
    '70 %'
  );
  // Имена параметров
  ValueNames: array[0..6] of String = (
    'Имя изделия',
    'Комментарий',
    'Материал',
    'Цвет изделия',
    'Дата изготовления',
    'Телефон склада',
    'Процесс доставки'
  );
```

Всё, теперь можно приступать к написанию кода, реализующего методы нашего класса. По ходу работы Я старался давать максимально полные и понятные комментарии к каждому действию.

```
//---------------------------------------------------------------------------
{* * * * * * * * * * * * * * * * * * TVTCustomEditor * * * * * * * * * * * * * * * * * *}
//---------------------------------------------------------------------------
destructor TVTCustomEditor.Destroy;
begin
  FreeAndNil(FEdit);
  inherited;
end;
```

```
//-----------------------------------------------------------------------------
// Для обработки нажатий с клавиатуры.
// Отмена редактирования по Escape, завершение редактирования по Enter,
// и переход между узлами по Up/Down, если список элементов у комбо бокса или
// редактора даты не выпущен.
//-----------------------------------------------------------------------------
procedure TVTCustomEditor.EditKeyDown(Sender: TObject; var Key: Word; Shift:
TShiftState);
var
  CanContinue: Boolean;
begin
  CanContinue := True;
  case Key of
    VK_ESCAPE: // Нажали Escape
      if CanContinue then
      begin
        FTree.CancelEditNode;
        Key := 0;
      end;
    VK_RETURN: // Нажали Enter
      if CanContinue then
      begin
        // Если Ctrl для TMemo не зажат, то завершаем редактирование
        // Сделаем так, чтобы можно было по Ctrl+Enter вставлять в Memo
        // новую линию.
        if (FEdit is TMemo) and (Shift = []) then
          FTree.EndEditNode
        else if not(FEdit is TMemo) then
          FTree.EndEditNode;
        Key := 0;
      end;
    VK_UP, VK_DOWN:
      begin
        // Проверить, не идёт ли работа с редактором. Если идёт, то запретить
        // активность дерева, если нет, то передать нажатие дереву.
        CanContinue := Shift = [];
        if FEdit is TComboBox then
          CanContinue := CanContinue and not TComboBox(FEdit).DroppedDown;
        if FEdit is TDateTimePicker then
          CanContinue := CanContinue and not
TDateTimePicker(FEdit).DroppedDown;
        if CanContinue then
        begin
          // Передача клавиши дереву
          PostMessage(FTree.Handle, WM_KEYDOWN, Key, 0);
          Key := 0;
        end;
      end;
  end;
end;

//-----------------------------------------------------------------------------
// Началось редактирование, нужно показать редактор и установить ему фокус.
//-----------------------------------------------------------------------------
function TVTCustomEditor.BeginEdit: Boolean;
begin
  Result := True;
  with FEdit do
  begin
    Show;
    SetFocus;
  end;
```

```
end;

//----------------------------------------------------------------------------
// Отменилось, прячем редактор.
//----------------------------------------------------------------------------
function TVTCustomEditor.CancelEdit: Boolean;
begin
  Result := True;
  FEdit.Hide;
end;

//----------------------------------------------------------------------------
// Успешно завершилось, прячем редактор, обновляем данные узла и возвращаем
// фокус дереву.
//----------------------------------------------------------------------------
function TVTCustomEditor.EndEdit: Boolean;
var
  Txt: String;
begin
  Result := True;
  if FEdit is TEdit then
    Txt := TEdit(FEdit).Text
  else if FEdit is TMemo then
    Txt := TEdit(FEdit).Text
  else if FEdit is TComboBox then
    Txt := TComboBox(FEdit).Text
  else if FEdit is TColorBox then
    Txt := ColorToString(TColorBox(FEdit).Selected)
  else if FEdit is TDateTimePicker then
  begin
    Txt := DateToStr(TDateTimePicker(FEdit).DateTime);
  end
  else if FEdit is TMaskEdit then
    Txt := TMaskEdit(FEdit).Text
  else if FEdit is TProgressBar then
    Txt := IntToStr(TProgressBar(FEdit).Position) + ' %';
  // Изменяем текст узла (поле Value у TVTEditNode) через событие OnNewText
  // у дерева
  FTree.Text[FNode, FColumn] := Txt;
  FEdit.Hide;
  FTree.SetFocus;
end;

//----------------------------------------------------------------------------
// Возвращаем границы редактора.
//----------------------------------------------------------------------------
function TVTCustomEditor.GetBounds: TRect;
begin
  Result := FEdit.BoundsRect;
end;

//----------------------------------------------------------------------------
// В подготовке к редактированию мы должны создать экземпляр TWinControl
// нужного класса потомка в соответствии с полем Kind у TVTEditNode.
//----------------------------------------------------------------------------
function TVTCustomEditor.PrepareEdit(Tree: TBaseVirtualTree; Node:
PVirtualNode;
  Column: TColumnIndex): Boolean;
var
  VTEditNode: PVTEditNode;
begin
  Result := True;
```

```
  FTree := Tree as TVirtualStringTree;
  FNode := Node;
  FColumn := Column;
  FreeAndNil(FEdit);
  VTEditNode := FTree.GetNodeData(Node);
  case VTEditNode.Kind of
    ekString:
    begin
      FEdit := TEdit.Create(nil);
      with FEdit as TEdit do
      begin
        AutoSize := False;
        Visible := False;
        Parent := Tree;
        Text := VTEditNode.Value;
        OnKeyDown := EditKeyDown;
      end;
    end;
    ekMemo:
    begin
      FEdit := TMemo.Create(nil);
      with FEdit as TMemo do
      begin
        Visible := False;
        Parent := Tree;
        ScrollBars := ssVertical;
        Text := VTEditNode.Value;
        OnKeyDown := EditKeyDown;
      end;
    end;
    ekComboBox:
    begin
      FEdit := TComboBox.Create(nil);
      with FEdit as TComboBox do
      begin
        Visible := False;
        Parent := Tree;
        Text := VTEditNode.Value;
        Items.Add('Шерсть');
        Items.Add('Хлопок');
        Items.Add('Шёлк');
        Items.Add('Кожа');
        Items.Add('Велюр');
        OnKeyDown := EditKeyDown;
      end;
    end;
    ekColor:
    begin
      FEdit := TColorBox.Create(nil);
      with FEdit as TColorBox do
      begin
        Visible := False;
        Parent := Tree;
        Selected := StringToColor(VTEditNode.Value);
        Style := Style + [cbPrettyNames];
        OnKeyDown := EditKeyDown;
      end;
    end;
    ekMask:
    begin
      FEdit := TMaskEdit.Create(nil);
      with FEdit as TMaskEdit do
```

```
      begin
        AutoSize := False;
        Visible := False;
        Parent := Tree;
        EditMask := '999-99-99';
        Text := VTEditNode.Value;
        OnKeyDown := EditKeyDown;
      end;
    end;
    ekDate:
    begin
      FEdit := TDateTimePicker.Create(nil);
      with FEdit as TDateTimePicker do
      begin
        Visible := False;
        Parent := Tree;
        Date := StrToDate(VTEditNode.Value);
        OnKeyDown := EditKeyDown;
      end;
    end;
    ekProgress:
    begin
      FEdit := TProgressBar.Create(nil);
      with FEdit as TProgressBar do
      begin
        Visible := False;
        Parent := Tree;
        Position := StrToIntDef(VTEditNode.Value, 70);
      end;
    end
  else
    Result := False;
  end;
end;

//-----------------------------------------------------------------------------
// Обработка сообщений Windows для редактора.
//-----------------------------------------------------------------------------
procedure TVTCustomEditor.ProcessMessage(var Message: TMessage);
begin
  FEdit.WindowProc(Message);
end;

//-----------------------------------------------------------------------------
// Устанавливает границы редактора в соответствии с шириной и высотой
// колонки.
//-----------------------------------------------------------------------------
procedure TVTCustomEditor.SetBounds(R: TRect);
var
  Dummy: Integer;
begin
  FTree.Header.Columns.GetColumnBounds(FColumn, Dummy, R.Right);
  FEdit.BoundsRect := R;
  if FEdit is TMemo then
    FEdit.Height := 80;
end;
```

Всё! Основную работу мы проделали. Настало время соединить наш класс с VT. Делается это передачей интерфейса в событии **OnCreateEditor**:

```
procedure TfrmMain.VTCreateEditor(Sender: TBaseVirtualTree; Node:
PVirtualNode;
  Column: TColumnIndex; out EditLink: IVTEditLink);
begin
  EditLink := TVTCustomEditor.Create;
end;
```

Суть его в простой передаче экземпляра нашего редактора. Теперь при начале редактирования VT будет вызывать методы нашего класса вместо своего стандартного редактора. Не забудьте обработать изменившиеся поля дерева:

```
procedure TfrmMain.VTNewText(Sender: TBaseVirtualTree; Node: PVirtualNode;
  Column: TColumnIndex; NewText: WideString);
var
  VTEditNode: PVTEditNode;
begin
  VTEditNode := Sender.GetNodeData(Node);
  with VTEditNode^ do
  begin
    if not Changed then
      Changed := Value <> NewText;
    Value := StringReplace(NewText, #13#10, ' ', [rfReplaceAll]);
  end;
end;
```

Редактирование само по себе не включится. В событии **OnEditing** необходимо разрешить его:

```
procedure TfrmMain.VTEditing(Sender: TBaseVirtualTree; Node: PVirtualNode;
  Column: TColumnIndex; var Allowed: Boolean);
begin
  Allowed := Column > 0;
end;
```

Об остальных событиях Вы можете сами позаботиться, благо знаний у Вас уже достаточно. Включите свойство TreeOptions.MiscOptions.toEditable у VT и запустите скомпилированное приложение. Каждый из семи узлов будет обладать собственным редактором.

Приведённый пример предоставляет достаточный запас информации для начала написания собственных элементов управления и внедрения их в VT. Вы имеете мощный инструмент для редактирования узлов, границы которого ограничиваются лишь вашей фантазией. В качестве домашнего задания можете попробовать связать VT с числовым редактором.

Проект Figure 1.4, демонстрирующий работу со специализированными редакторами данных находится в папке Fig 1.4.

5.0. Полное изменение отрисовки дерева. Класс TVirtualDrawTree.

В этой главе мы с Вами рассмотрим работу с классом TVirtualDrawTree, специально предназначенного для отрисовки нестандартных в отображении узлов. Он не ориентирован на отображение каких-то специфических данных, вроде текста. TVirtualDrawTree специально предназначен исключительно для отрисовки различных элементов дизайна вашей программы. В нашем примере мы рассмотрим создание полностью видоизменённого дерева, напоминающего собой менеджер закачек Mozilla Firefox с отображением различных элементов управления. Но для начала разберёмся в отличиях TVirtualDrawTree от TVirtualStringTree, с которым мы ранее работали.

5.1. Отличительные черты класса TVirtualDrawTree.

Различия минимальны. Сразу бросается в глаза отсутствие привычных событий OnGetText и OnNewText, что говорит о непредназначенности дерева для работы с текстовыми данными. Тем не менее, Вы по-прежнему можете использовать собственные редакторы, всё для этого на месте. Появилось целых четыре новых события:

- **OnDrawNode** – если у TVirtualStringTree основа – событие OnGetText, то у TVirtualDrawTree это OnDrawNode. В этом событии необходимо будет полностью реализовать всю отрисовку узла.
- **OnGetNodeWidth** – предназначено для указания ширины узла. Напомню, что высота узла указывается в параметре DefaultNodeHeight для деревьев с узлами одинаковой высоты или в событии OnMeasureItem для деревьев с узлами различной высоты.
- **OnGetHintSize** – предназначено для указания размеров всплывающей подсказки.
- **OnDrawHint** – событие, позволяющее отрисовку всплывающей подсказки. До сих пор мы могли лишь указать текст подсказки, но с этого момента мы должны будем заниматься полным отображением подсказки самостоятельно.

Всё остальное не претерпело никаких изменений.

5.2. Самостоятельное отображение узлов и их подсказок.

В тексте данной главы Я ограничился лишь указанием примерной реализации событий OnDrawNode и OnDrawHint, а также реализацией потока. Всё остальное в рамках данной главы довольно объёмно и не представляет для нас большого интереса, поэтому, чтобы увидеть результат работы наших трудов, необходимо скомпилировать пример из папки Fig 1.5.

Предположим, что нам необходимо создать небольшую базу данных заказов продуктового магазина. Мы должны предусмотреть отображение картинок изделий, их названий, цен и т.д. Также, нам необходимо визуализировать процесс доставки какого-то заказа. Задание нетривиальное, но вполне реализуемое с помощью TVirtualDrawTree.

Как обычно, начинаем с продумывания структуры данных каждого узла:

```
type
  PItemNode = ^TItemNode;

  TItemNode = record
    Image, Name: String;
    Mass, PriceKg: Word;
    Process: Byte;
    PersonalThread: Cardinal;
  end;
```

Нам необходимо хранить путь к картинке, имя изделия, вес и цену за килограмм. Поле Process необходимо для имитации процесса доставки. Его значение мы будем изменять с помощью отдельного потока, а отображать с помощью TProgressBar в выделенном узле. Хендл потока будет храниться в поле PersonalThread.

Переходим к реализации событий:

```
//----------------------------------------------------------------------------
// Отрисовка всплывающей подсказки.
```

```
//---------------------------------------------------------------------------
procedure TfrmMain.VTDrawHint(Sender: TBaseVirtualTree; HintCanvas: TCanvas;
  Node: PVirtualNode; R: TRect; Column: TColumnIndex);
var
  NodeData: PItemNode;
  Th: Integer;
  ImgRect, RealRect, NameRect, PriceRect: TRect;
  Img: TPicture;
begin
  // Области отрисовки различных элементов
  Th := HintCanvas.TextHeight('Wj');
  RealRect := R;
  // Границы картинки
  ImgRect := RealRect;
  with ImgRect do
  begin
    Inc(Left, 16);
    Inc(Top, 8);
    Right := Left + 64;
    Bottom := Top + 64;
  end;
  // Границы текста наименования
  NameRect := RealRect;
  with NameRect do
  begin
    Inc(Left, 96);
    Inc(Top, 8);
    Bottom := Top + Th;
  end;
  // Границы текста общей цены
  PriceRect := RealRect;
  with PriceRect do
  begin
    Inc(Left, 96);
    Dec(Bottom, 16);
    Top := Bottom - Th;
  end;
  // Рисуем фон
  with HintCanvas do
  begin
    with Brush do
    begin
      Color := clInfoBk;
      Style := bsSolid;
    end;
    Pen.Color := clBlack;
    Pen.Width := 1;
    // Жёлтый фон и чёрная рамка по бокам
    Rectangle(RealRect);
  end;
  // Текст
  NodeData := Sender.GetNodeData(Node);
  with HintCanvas do
  begin
    Brush.Style := bsClear;
    Font.Color := clWindowText;
    // Наименование
    Font.Style := [fsBold];
    if NodeData^.Process = 0 then
      TextOut(NameRect.Left, NameRect.Top, NodeData^.Name + ' (не
доставлено)')
    else if (NodeData^.Process > 0) and (NodeData^.Process < 100) then
```

```
      TextOut(NameRect.Left, NameRect.Top, NodeData^.Name + '
(доставляется)')
    else if NodeData^.Process = 100 then
      TextOut(NameRect.Left, NameRect.Top, NodeData^.Name + ' (доставлено)');
    // Общая цена
    Font.Style := [];
    TextOut(PriceRect.Left, PriceRect.Top, 'Общая цена: ' +
IntToStr(NodeData^.PriceKg * NodeData^.Mass) + ' р.');
  end;
  // Картинка
  Img := TPicture.Create;
  try
    Img.LoadFromFile(NodeData^.Image);
    HintCanvas.Draw(ImgRect.Left, ImgRect.Top, Img.Graphic);
    with HintCanvas, ImgRect do
    begin
      Pen.Width := 2;
      Pen.Color := $00B99D7F;
      Rectangle(Rect(ImgRect.Left, Top, Left + Img.Width, Top + Img.Height));
    end;
  finally
    FreeAndNil(Img);
  end;
  // Отображаем процесс выполнения
  with FProgress do
  begin
    Position := NodeData^.Process;
    Width := RealRect.Right - 96 - 16;
    PaintTo(HintCanvas, NameRect.Left, NameRect.Top + Th + Th div 2);
  end;
end;

//----------------------------------------------------------------------------
// Отрисовка узла.
//----------------------------------------------------------------------------
procedure TfrmMain.VTDrawNode(Sender: TBaseVirtualTree;
  const PaintInfo: TVTPaintInfo);
var
  NodeData: PItemNode;
  Th: Integer;
  ImgRect, RealRect, NameRect, GetRect, CancelRect, MassRect, PriceRect:
TRect;
  Img: TPicture;
begin
  // Области отрисовки различных элементов
  Th := PaintInfo.Canvas.TextHeight('Wj');
  RealRect := PaintInfo.CellRect;
  ImgRect := RealRect;
  with ImgRect do
  begin
    Inc(Left, 16);
    Inc(Top, 16);
    Right := Left + 64;
    Bottom := Top + 64;
  end;
  NameRect := RealRect;
  with NameRect do
  begin
    Inc(Left, 96);
    Inc(Top, 16);
    Bottom := Top + Th;
  end;
```

```
  GetRect := RealRect;
  with GetRect do
  begin
    Dec(Bottom, 16 + Th);
    Top := Bottom - Th;
    Dec(Right, 16);
    Left := Right - PaintInfo.Canvas.TextWidth('Доставить');
  end;
  CancelRect := GetRect;
  with CancelRect do
  begin
    Inc(Top, Th);
    Inc(Bottom, Th);
  end;
  MassRect := RealRect;
  with MassRect do
  begin
    Inc(Left, 96);
    Dec(Bottom, 16 + Th);
    Top := Bottom - Th;
  end;
  PriceRect := MassRect;
  with PriceRect do
  begin
    Inc(Top, Th);
    Inc(Bottom, Th);
  end;
  // Рисуем фон
  with PaintInfo.Canvas do
  begin
    with Brush do
    begin
      Color := clWindow;
      Style := bsSolid;
    end;
    Pen.Color := $00B99D7F;
  end;
  if PaintInfo.Node = Sender.FocusedNode then
    GradFill(PaintInfo.Canvas.Handle, RealRect, $00EAE2D9, $00D7C8B7, gkVert)
  else
    PaintInfo.Canvas.FillRect(RealRect);
  // Текст
  NodeData := Sender.GetNodeData(PaintInfo.Node);
  with PaintInfo.Canvas do
  begin
    Brush.Style := bsClear;
    Font.Color := clWindowText;
    // Наименование
    Font.Style := [fsBold];
    if NodeData^.Process = 0 then
      TextOut(NameRect.Left, NameRect.Top, NodeData^.Name + ' (не
доставлено)')
    else if (NodeData^.Process > 0) and (NodeData^.Process < 100) then
      TextOut(NameRect.Left, NameRect.Top, NodeData^.Name + '
(доставляется)')
    else if NodeData^.Process = 100 then
      TextOut(NameRect.Left, NameRect.Top, NodeData^.Name + ' (доставлено)');
    // Масса
    Font.Style := [];
    TextOut(MassRect.Left, MassRect.Top, 'Масса: ' + IntToStr(NodeData^.Mass)
+ ' кг.');
    // Цена
```

```
    TextOut(PriceRect.Left, PriceRect.Top, 'Цена за килограмм: ' +
IntToStr(NodeData^.PriceKg) + ' р.');
    // Доставить
    with Font do
    begin
      Style := [fsUnderline];
      Color := clBlue;
    end;
    TextOut(GetRect.Left, GetRect.Top, 'Доставить');
    // Отмена
    TextOut(CancelRect.Left, CancelRect.Top, 'Отмена');
  end;
  // Картинка
  Img := TPicture.Create;
  try
    Img.LoadFromFile(NodeData^.Image);
    PaintInfo.Canvas.Draw(ImgRect.Left, ImgRect.Top, Img.Graphic);
    if PaintInfo.Node = Sender.FocusedNode then
      with PaintInfo.Canvas, ImgRect do
      begin
        Pen.Width := 2;
        Rectangle(Rect(ImgRect.Left, Top, Left + Img.Width, Top +
Img.Height));
      end;
  finally
    FreeAndNil(Img);
  end;
  // Процесс
  if PaintInfo.Node = Sender.FocusedNode then
    with FProgress do
    begin
      Position := NodeData^.Process;
      Width := RealRect.Right - 96 - 16;
      PaintTo(PaintInfo.Canvas, NameRect.Left, NameRect.Top + Th + Th div 2);
    end;
end;

//---------------------------------------------------------------------------
// Метод предназначен для отображения курсора руки над надписями "Доставить"
// и "Отмена", которые нарисованы в стиле ссылок.
//---------------------------------------------------------------------------
procedure TfrmMain.VTGetCursor(Sender: TBaseVirtualTree; var Cursor:
TCursor);
var
  Pt: TPoint;
  Node: PVirtualNode;
  NRect: TRect;
  D: Integer;
  Inf: tagScrollInfo;
begin
  GetCursorPos(Pt);
  Pt := Sender.ScreenToClient(Pt);
  Node := Sender.GetNodeAt(Pt.X, Pt.Y);
  with Inf do
  begin
    cbSize := SizeOf(tagScrollInfo);
    fMask := SIF_RANGE;
  end;
  GetScrollInfo(Sender.Handle, SB_VERT, Inf);
  if (Inf.nMax > Sender.Height) then
    D := GetSystemMetrics(SM_CXHTHUMB)
  else
```

```
    D := 0;
  if Node <> nil then
  begin
    NRect := Sender.GetDisplayRect(Node, -1, False);
    if ((Pt.X > Sender.Width - 71 - D) and (Pt.X < Sender.Width - 16 - D) and
      (Pt.Y > NRect.Bottom - 42) and
      (Pt.Y < NRect.Bottom - 29))
      or
      ((Pt.X > Sender.Width - 71 - D) and (Pt.X < Sender.Width - 35 - D) and
      (Pt.Y > NRect.Bottom - 29) and
      (Pt.Y < NRect.Bottom - 16))
    then
      Cursor := crHandPoint
    else
      Cursor := crDefault;
  end;
end;

//---------------------------------------------------------------------------

procedure TfrmMain.VTGetHintSize(Sender: TBaseVirtualTree; Node:
PVirtualNode;
  Column: TColumnIndex; var R: TRect);
begin
  with R do
  begin
    Left := 0;
    Top := 0;
    Right := 250;
    Bottom := 80;
  end;
end;

//---------------------------------------------------------------------------
// Получение ширины узла. Если Вы включили опцию toFullRowSelect, то
// в контексте данного примера это событие необязательно.
//---------------------------------------------------------------------------
procedure TfrmMain.VTGetNodeWidth(Sender: TBaseVirtualTree; HintCanvas:
TCanvas;
  Node: PVirtualNode; Column: TColumnIndex; var NodeWidth: Integer);
var
  D: Integer;
  Inf: tagScrollInfo;
begin
  // Смотрим, виден ли вертикальный скрол.
  // Если есть, то его ширину нужно будет вычесть из ширины узла
  with Inf do
  begin
    cbSize := SizeOf(tagScrollInfo);
    fMask := SIF_RANGE;
  end;
  GetScrollInfo(Sender.Handle, SB_VERT, Inf);
  if (Inf.nMax > Sender.Height) then
    D := GetSystemMetrics(SM_CXHTHUMB)
  else
    D := 0;
  // Вычитаем из ширины дерева ширину вертикального скрола, отступа и края
  // дерева
  NodeWidth := Sender.Width - D - VT.Indent - VT.Margin;
end;
```

Реализация потока, имитирующего процесс доставки, будет выглядеть так (комментарии даны по ходу листинга):

```
// Поток, имитирующий выполнение какой-то операции
TProcessThread = class(TThread)
private
  FWnd: HWND; // Хендл формы. Необходим для отсылки сообщений ему
  FNode: PVirtualNode; // Узел, за который поток отвечает
  FValue: Byte; // Процесс доставки
  procedure RepaintTree;
  procedure CheckDeleted;
public
  constructor Create(Suspended: Boolean; var InitialValue: Byte;
    CallingForm: HWND; VTNode: PVirtualNode);
protected
  procedure Execute; override;
end;

constructor TProcessThread.Create(Suspended: Boolean;
  var InitialValue: Byte; CallingForm: HWND; VTNode: PVirtualNode);
begin
  inherited Create(Suspended);
  FWnd := CallingForm;
  FNode := VTNode;
  FValue := InitialValue;
  FreeOnTerminate := True;
end;

//---------------------------------------------------------------------------
// Отсылаем сообщение форме о том, что узлу FNode необходима перерисовка
// Параметры:
//   WParam - адрес узла;
//   LParam - текущее значение процесса доставки.
//---------------------------------------------------------------------------
procedure TProcessThread.RepaintTree;
begin
  SendMessage(FWnd, VT_REPAINTTREE, Integer(FNode), FValue);
end;

//---------------------------------------------------------------------------
// Проверить, не был ли удалён узел, за который мы отвечает. Если был, то
// делать нам больше нечего.
//---------------------------------------------------------------------------
procedure TProcessThread.CheckDeleted;
begin
  if not Assigned(FNode) then
    Terminate;
end;

//---------------------------------------------------------------------------

procedure TProcessThread.Execute;
begin
  while FValue < 100 do
  begin
    Sleep(100);
    Synchronize(CheckDeleted);
    Inc(FValue);
    Synchronize(RepaintTree);
  end;
end;
```

Обработка сообщения VT_REPAINTTREE у формы:

```
const
  // Сообщение для уведомления дерева потоком о том, что необходимо
  // перерисовать указанный узел
  VT_REPAINTTREE = WM_USER + 500;

procedure TfrmMain.VTREPAINTTREE(var Message: TMessage);
var
  Node: PVirtualNode;
  ItemNode: PItemNode;
begin
  Node := Pointer(Message.WParam);
  if Assigned(Node) then
  begin
    ItemNode := VT.GetNodeData(Node);
    ItemNode^.Process := Message.LParam;
    VT.RepaintNode(Node);
  end;
end;
```

Для правильной работы примера необходимо настроить дерево. Чтобы включить подсказку, необходимо изменить свойство **HintMode** на **hmHint** (это свойство будет подробнее рассмотрено позже).

Теоритически, наше задание можно было бы выполнить и с помощью обычного TVirtualStringTree, но он ориентирован только на отображение текстовых данных и неизвестно какие проблемы и опасности могут поджидать Вас при работе с ним. Просто запомните, если Вам требуется существенное изменение отображения данных, то используйте TVirtualDrawTree, в противном случае достаточно TVirtualStringTree.

Полная версия проекта Figure 1.5, демонстрирующего изменённую отрисовку дерева с использованием TVirtualDrawTree, находится в папке Fig 1.5. В нём также присутствуют дополнительные комментарии к коду.

6.0. Сохранение и загрузка VT из файла.

Какой толк от данных, если результат вычислений нигде не хранить? Давайте посмотрим что есть в VT для обеспечения сохранения и загрузки его содержимого и какими способами это можно сделать. Мы возьмем максимально простой пример: текстовое дерево из двух колонок с возможной вложенностью узлов. Такая картина наиболее часто встречается в повседневной жизни.

6.1. Хранение дерева стандартными средствами.

VT реализует такие методы, как LoadFromFile и SaveToFile, предназначенные для загрузки и сохранения дерева в файл. Основа их действия - методы LoadFromStream и SaveToStream, предназначенные для работы с потоком. Давайте попробуем вызвать их и проанализировать полученный файл.

Данные для узла:

```
type
  PNodeData = ^TNodeData;

  TNodeData = record
    Caption: WideString;
    Value: Integer;
  end;
```

Заполняем дерево случайными данными:

```
const
  Strs: array[0..4] of WideString = (
    'Εκλογές στο Ισραήλ',
    'Απεβίωσε ο Γιάννης Ξενάκης',
    'Μεταμόσχευση χεριού',
    'Σάμπα, κλόουν, μασκαράδες και πενιές',
    'Δοκιμάζει τις αντοχές του'
  );
  Ints: array[0..4] of Integer = (
    938,
    5346,
    23,
    789,
    2888
  );

procedure TfrmMain.BtnFillClick(Sender: TObject);
var
  i, j: Integer;
  NewNode1, NewNode2: PVirtualNode;
  ItemData: PNodeData;
begin
  Randomize;
  for i := 0 to 1 do
  begin
    NewNode1 := VT.AddChild(nil);
    ItemData := VT.GetNodeData(NewNode1);
    with ItemData^ do
    begin
      Caption := Strs[Random(4)];
      Value := Ints[Random(4)];
    end;
    for j := 0 to 9 do
    begin
      NewNode2 := VT.AddChild(NewNode1);
      ItemData := VT.GetNodeData(NewNode2);
```

```
      with ItemData^ do
      begin
        Caption := Strs[Random(4)];
        Value := Ints[Random(4)];
      end;
    end;
  end;
end;
```

Попробуем без лишних телодвижений просто вызвать оба метода и посмотреть на результат:

```
procedure TfrmMain.BtnLoadClick(Sender: TObject);
begin
  VT.LoadFromFile('vt.dat');
end;

procedure TfrmMain.BtnSaveClick(Sender: TObject);
begin
  VT.SaveToFile('vt.dat');
end;
```

Добавьте обработчик OnGetText дереву для отображения текста и скомпилируйте пример. Нажмите на кнопку, которой назначен обработчик BtnFillClick, чтобы заполнить дерево. Затем жмите на кнопку сохранения. В папке с примером появится файл vt.dat, откроем его блокнотом или любым hex редактором. Что мы видим? Бинарный файл, в котором проскакивает текстовая информация, очевидно взятая из поля Caption нашей структуры. Хорошо, очистите дерево и попробуйте загрузить его, нажав на кнопку загрузки. Структура дерева восстановится, но все его узлы будут пусты… Зачем же тогда в полученном файле присутствует содержимое узлов? Дерево сохраняет содержимое узлов, но при этом никак не загружает его. Получается, что это мусор в файле? И да и нет. Такое поведение дерева объясняется тем, что текст узлов (но не данные узла) помещается в поток TStream при операциях с буфером обмена и drag&drop. Затем он читается из потока для восстановления узла. Но нужно это только для передачи текста узлов сторонним приложениям и контроллам, например TRichEdit или MSWord (они же ничего про данные узла знать не могут), а в нашем случае мы работаем с файлом, поэтому эта информация будет проигнорирована деревом, как информация, не предназначенная для загрузки из файла.

Но что тогда отвечает за процесс загрузки и сохранения? Как избавиться от этого «мусора» в файле? Для начала, запретим дереву помещать эту информацию в поток. За это отвечает опция **toSaveCaptons** из StringOptions раздела настроек дерева, которую необходимо в нашем случае поставить в False. Теперь необходимо самостоятельно реализовать запись содержимого узла в поток. Для этого нам в помощь присутствуют два простых события:

- **OnLoadNode** - вызывается при чтении данных для каждого узла.
- **OnSaveNode** - вызывается при сохранении данных для каждого узла.

Таким образом, наша задача заключается только в заполнении содержимого потока по мере записи в него структуры дерева. О записи структуры дерева VT заботится самостоятельно.

Добавим обработчики обоим событиям:

```
procedure TfrmMain.VTLoadNode(Sender: TBaseVirtualTree; Node: PVirtualNode;
  Stream: TStream);
var
  Reader: TReader;
  NodeData: PNodeData;
begin
  Reader := TReader.Create(Stream, 8096);
  try
    NodeData := Sender.GetNodeData(Node);
```

```
    with NodeData^, Reader do
    begin
      Caption := ReadWideString;
      Value := ReadInteger;
    end;
  finally
    FreeAndNil(Reader);
  end;
end;

procedure TfrmMain.VTSaveNode(Sender: TBaseVirtualTree; Node: PVirtualNode;
  Stream: TStream);
var
  Writer: TWriter;
  NodeData: PNodeData;
begin
  Writer := TWriter.Create(Stream, 8096);
  try
    NodeData := Sender.GetNodeData(Node);
    with Writer, NodeData^ do
    begin
      WriteWideString(Caption);
      WriteInteger(Value);
    end;
  finally
    FreeAndNil(Writer);
  end;
end;
```

Тем, кто работал с массивами и реализовывал их методы загрузки и сохранения, этот код покажется довольно знакомым. Мы используем классы TReader и TWriter для работы с потоком. В нашем случае мы помещаем в поток строку и число от каждого узла. При загрузке они также легко извлекаются из потока. Снова скомпилируйте пример и повторите предыдущие действия по сохранению дерева в файл. На этот раз мы увидим иную картину. В файле уже не будет прежнего мусора, а строки будут сохранены в Юникоде и завершены null-terminating (#0) символом. По соседству с ними будут располагаться числа. Вернёмся в приложение. Очистите дерево и нажмите на кнопку загрузки. Вуаля! Дерево восстановлено в своём первозданном виде. Таким образом, Вы можете хранить любые типы данных (кроме referenced естественно) в бинарном файле и легко их загружать.

6.2. Хранение дерева в XML файле.

Но что если Вас не устраивает перспектива хранения дерева в бинарном файле? Что если у Вас есть собственный формат данных, который Вы хотите использовать? В этом случае все действия по загрузке и сохранения можно полностью выполнить самому. Вот простой пример как можно то же самое дерево из предыдущего примера хранить в XML файле. Для работы с XML Я использовал стандартный парсёр от Microsoft (TXMLDocument). Тем не менее, Вам ничто не мешает переписать пример под Ваш любимый парсёр.

Зарузка:

```
procedure TfrmMain.BtnLoadXMLClick(Sender: TObject);
var
  XMLDocument: TXMLDocument;

  procedure LoadXML(const ANodeList: IXMLNodeList; AParent: PVirtualNode);
  var
    i: Integer;
    NewNode: PVirtualNode;
    NodeData: PNodeData;
  begin
```

```
    for i := 0 to ANodeList.Count - 1 do
    begin
      NewNode := VT.AddChild(AParent);
      NodeData := VT.GetNodeData(NewNode);
      with NodeData^ do
      begin
        Caption := VarToWideStr(ANodeList[i].Attributes['Caption']);
        Value := StrToIntDef(
                VarToStr(
                ANodeList[i].Attributes['Value']
                ), 0);
      end;
      LoadXML(ANodeList[i].ChildNodes, NewNode);
    end;
  end;

begin
  if not FileExists('vt.xml') then
  begin
    MessageBox(Handle, 'Файл vt.xml не найден.', PChar(Application.Title),
      MB_ICONINFORMATION + MB_OK);
    Exit;
  end;
  XMLDocument := TXMLDocument.Create(Self);
  try
    XMLDocument.LoadFromFile('vt.xml');
    // Для каждой ветки XML дерева создать узел в дереве и зарузить поля
    // для структуры данных из аттрибутов
    // Не забываем использовать блоки Begin/End Update.
    VT.BeginUpdate;
    try
      LoadXML(XMLDocument.DocumentElement.ChildNodes, nil);
    finally
      VT.EndUpdate;
    end;
  finally
    FreeAndNil(XMLDocument);
  end;
end;
```

Сохранение:

```
procedure TfrmMain.BtnSaveXMLClick(Sender: TObject);
var
  XMLDocument: TXMLDocument;

  procedure SaveXML(ANode: PVirtualNode; const AParent: IXMLNode);
  var
    i: Integer;
    NewNode: IXMLNode;
    NextNode: PVirtualNode;
    NodeData: PNodeData;
  begin
    NextNode := ANode.FirstChild;
    if Assigned(NextNode) then
      repeat
        NodeData := VT.GetNodeData(NextNode);
        NewNode := AParent.AddChild('Node');
        with NewNode, NodeData^ do
        begin
```

```
          Attributes['Caption'] := Caption;
          Attributes['Value'] := Value;
        end;
        SaveXML(NextNode, NewNode);
        NextNode := NextNode.NextSibling;
      until
        NextNode = nil;
  end;

begin
  XMLDocument := TXMLDocument.Create(Self);
  try
    with XMLDocument do
    begin
      Active := True;
      Encoding := 'UTF-16';
      AddChild('VirtualTreeview');
      Options := Options + [doNodeAutoIndent];
    end;
    SaveXML(VT.RootNode, XMLDocument.DocumentElement);
    XMLDocument.SaveToFile('vt.xml');
  finally
    FreeAndNil(XMLDocument);
  end;
end;
```

6.3. Экспорт содержимого дерева в различные форматы.

VT предлагает замечательную возможность экспортировать своё содержимое в различные форматы данных. На данный момент поддерживается текст, Юникод, RTF и HTML. Что примечательно, VT экспортирует содержимое в текст, сохраняя при этом табличную структуру. Вот короткий пример экспорта в HTML с сохранением в файл. Экспорт для всех остальных форматов проходит примерно также.

```
procedure TfrmMain.BtnExportHTMLClick(Sender: TObject);
const
  HTMLHead = '<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0//EN">'#13#10 +
    '<html>'#13#10 +
    '<head>'#13#10 +
    '  <meta http-equiv="Content-Type" content="text/html; charset=utf-
8">'#13#10 +
    '  <title>Virtual Treeview HTML</title>'#13#10 +
    '</head>'#13#10 +
    '<body>'#13#10;
  HTMLFoot = '</body>'#13#10 +
    '</html>' + #13#10;
var
  Fs: TFileStream;
  Str: String;
  Data: Pointer;
begin
  Fs := TFileStream.Create('html.html', fmCreate);
  try
    Str := HTMLHead + VT.ContentToHTML(tstAll) + HTMLFoot;
    Data := PChar(Str);
    Fs.WriteBuffer(Data^, Length(Str));
  finally
    FreeAndNil(fs);
  end;
  ShellExecute(Handle, 'open', 'html.html', nil, nil, SW_RESTORE);
end;
```

Пример Figure 1.6, демонстрирующий загрузку и сохранеие дерева находится в папке Fig 1.6.

7.0. Обзор типа TVirtualNode.

До сих пор наша работа с VT неизменно была связана с узлами, представленными типом TVirtualNode, но мы не уделяли ему особого внимания. А ведь он содержит много полезных и интересных свойств, например, позволяет задать тип отметки для конкретных узлов и получить доступ ко многим его свойствам. Восполним этот пробел.

7.1. Стандартные свойства.

Каждый узел, являясь экземпляром типа TVirtualNode, содержит множетсво вспомогательных и информативных свойств, которые с успехом можно использовать ради любых целей.

- **Index** – индекс узла в дереве по отношении к своему родителю. Т.е. абсолютным индексом он не является.
- **ChildCount** – количество дочерних элементов узла. Вручную не изменяется. Заполняется в обработчике события **OnInitChildren**.
- **NodeHeight** – высота узла. VT позволяет узлам иметь различные размеры, применение этой возможности будет подробнее рассмотрено ниже. Фактически, это read only поле, т.к. его изменение, кроме как к глюкам отрисовки, ни к чему не приводит. Корректное изменение этого свойства возможно только в событии **OnMeasureItem**. Если планируете использование этого поля, то желательно включить TreeOptions.MiscOptions.toVariableNodeHeight опцию дерева.
- **States** – набор настроек узла.
  - **vsInitialized** – True, если данный узел прошёл событие OnInitNode.
  - **vsChecking** – True, если пользователь зажал отметку и ещё не отпустил кнопку мыши.
  - **vsCutOrCopy** – узел был определён для копирования или вставки.
  - **vsDisabled** – узел выключен и не отвечает на действия пользователя.
  - **vsDeleting** – устанавливается деревом в True сразу перед тем, как узел должен быть удалён.
  - **vsExpanded** – True, если узел раскрыт.
  - **vsHasChildren** – позволяет указать наличие дочерних узлов. При этом узел не обязательно должен их иметь.
  - **vsVisible** – True, если узел видно в дереве.
  - **vsSelected** – True, если узел выделен.
  - **vsAllChildrenHidden** – True, если у узла включено свойство vsHasChildren, но все его дочерние элементы спрятяны (имеют свойство vsVisible в True).
  - **vsClearing** – True, если дочерние узлы начали удаляться.
  - **vsMultiline** – True, если текст узла должен быть перенесён, чтобы совпасть с шириной колонки.
  - **vsHeightMeasured** – True, если высота узла была опередедена в обработчике события OnMeasureItem.
  - **vsToggling** – используется деревом для предотвращения зацикливания при сворачивании/разворачивания узла.
- **Align** – смещение узла в диапазоне от 0 до 255. Чем больше значение, тем ниже будут располагаться кнопка сворачивания/разворачивания и отметка. Данное поле тем не менее не влияет на положение текста в узле. Часто испольузтеся вместе со свойством дерева **NodeAlignment**, которое будет рассмотрено ниже. Возможно установить только в событии инициализации.
- **CheckState** – содержит текущее состояние отметки.
  - **csUncheckedNormal** – узел не отмечен и не зажат.
  - **csUncheckedPressed** – узел не отмечен, но пользователь зажал кнопку мыши над отметкой.
  - **csCheckedNormal** – узел отмечен.
  - **csCheckedPressed** – узел зажат, но пользователь зажал кнопку мыши над отметкой.
  - **csMixedNormal** – толко для отметок типа TCheckBox в узлах со свойством ctTriStateCheckBox. Обозначает, что отметка имеет промежуточное состояние (аналог Grayed в TCheckBox).
  - **csMixedPressed** – то же самое, но отметка зажата левой кнопкой мыши.
- **CheckType** – тип отметки. Возможно установить только в событии инициализации.
  - **ctNone** – узел без отметки.

  - **ctTriStateCheckBox** - узел с отметкой типа TCheckBox, которая может иметь 3 состояния (отмечена, grayed, не отмечена).
  - **ctCheckBox** - узел с отметкой типа TCheckBox, которая может иметь 2 состояния.
  - **ctRadioButton** - узел с отметкой типа TRadioButton.
  - **ctButton** - узел с кнопкой слева от надписи.
- **Dummy** - зарезервировано. Необходимо для выравнивания размера структуры узла до четырёх байтов (DWORD).
- **TotalCount** - сумма узла и всех его дочерних узлов и их дочерних узлов и т.д.
- **TotalHeight** - общая высота, занимаемая на экране данным узлом и всеми его дочерними узлами.
- **Parent** - указатель на узел родителя для данного узла.
- **PrevSibling** - указатель на предыдущий узел связанного списка. Если данный узел является первым в списке, то поле равно nil.
- **NextSibling** - то же самое, но указывает на следующий узел в списке.
- **FirstChild** - указатель на самый первый дочерний узел.
- **LastChild** - указатель на самый последний дочерний узел.

7.2. Отметки для узла.

Как Вы могли уже видеть выше, в отличие от TListView и TTreeView у VT есть аж четыре (!) вида отметки. Давайте подробнее рассмотрим каждый.

ctCheckBox - отметка типа TCheckBox, знакомая Вам по TListView и TTreeView. Один раз нажал - повесил галочку, другой раз нажал - снял галочку.

ctRadioButton - отметка типа TRadioButton. В списке узлов одновременно может быть отмечен только один узел.

ctTriStateCheckBox - это отметка наподобие TCheckBox, но включающая в себя также третье состояние (grayed), изображённое в виде квадрата, а не галки. Такой вид отметок часто можно встретить в приложениях, предлагающих выбрать какой-то набор компонетов для установки. В них отметка принимает grayed-вид, если выбрана только часть компонент из предложеннных для установки.

ctButton - немного необычный вид отметки. Очень удобно использовать для показа popup меню для индивидуального узла. Смотрится довольно оригинально и профессионально.

Учтите, что отметки будут видны только в главной колонке дерева, т.е. в той, что имеет индекс MainColumn.

Идеальное место для включения отметок - событие **OnInitNode**:

```
procedure TfrmMain.VTInitChildren(Sender: TBaseVirtualTree; Node:
PVirtualNode;
  var ChildCount: Cardinal);
begin
  if Sender.GetNodeLevel(Node) < 5 then
    ChildCount := 6
  else
    ChildCount := 0;
end;

procedure TfrmMain.VTInitNode(Sender: TBaseVirtualTree; ParentNode,
  Node: PVirtualNode; var InitialStates: TVirtualNodeInitStates);
var
  Lvl: Integer;
begin
  Lvl := Sender.GetNodeLevel(Node);
  if Lvl < 5 then
  begin
    InitialStates := InitialStates + [ivsHasChildren];
    case Lvl of
      0: Node.CheckType := ctButton;
      1: Node.CheckType := ctRadioButton;
      2: Node.CheckType := ctTriStateCheckBox;
      3: Node.CheckType := ctTriStateCheckBox;
      4: Node.CheckType := ctCheckBox;
      5: Node.CheckType := ctNone;
```

```
    end;
  end;
end;
```

В этом примере дерево будет заполняться шестью узлами для каждого родительского узла пока их уровень вложенности не достигнет шести. Тип отметки выбирается в соответствии с уровнем вложенности узла.

С отметками связаны события:

- **OnChecking** – происходит после нажатия на отметку и перед тем, как отпускается кнопка мыши.
- **OnChecked** – происходит сразу после того, как была отпущена кнопка мыши.

VT также позволяет указать изображения, которые будут использоваться для отметок. Кроме того, Вы можете нарисовать собственную картинку для каждого типа отметки и её состояний. VT уже имеет несколько готовых вариантов картинок для отметок, Вы можете просмотреть их в свойстве **CheckImageKind**. Чтобы использовать собственные изображения, поместите их в TImageList и присоедините к дереву через свойство CustomCheckImages. Переключите свойство CheckImageKind в ckCustom и отметки примут вид Ваших картинок. В папке Fig 1.7\res находится файл-пример, содержащий изменённые изображения отметок. Вы можете использовать его как шаблон для создания собственных изображений.

7.3. Опция toAutoTriStateTracking.

Если Вы уже скомпилировали пример и проверили его работу, то наверняка заметили, что у узлов с отметкой типа ctTriStateCheckBox состояние устанавливается автоматически в соответствии с состоянием отметки дочерних узлов. Если ни один из дочерних узлов не отмечен, то и данный узел не отмечен. Если все дочерние узлы отмечены, то и данный узел отмечен. Если только часть дочерних узлов отмечена, то данный узел будет иметь состояние отметки grayed (csMixedNormal). Надо отдать должное разработчику компонента за такую удобную возможность. Впрочем, если Вы не хотите избежать такого поведения дерева, то просто отключите опцию **toAutoTriStateTracking**, обитающую в TreeOptions.AutoOptions. Именно она отвечает за всё это.

Чтобы лучше закрепить знания о TVirtualNode типе, ознакомьтесь с примером Figure 1.7. В нём очень хорошо видна работа каждого свойства типа.

Проект Figure 1.7, демонстрирующий работу с типом TVirtualNode находится в папке Fig 1.7.

8.0. Небольшие примеры кода. Описание различных свойств VT, не попадающих под категории, рассмотренные выше.

Я специально решил выделить такие моменты в отдельную главу, чтобы не устраивать кашу среди предыдущих примеров и в то же время акцентировать внимание и на них. В конце статьи Я также полностью рассмотрю назначение всех опций VT.

8.1. Цветовые настройки.

VT выделяет все настраиваемые цвета (кроме **Color**) в отдельный небольшой класс - TVTColors, наследованный от TPersistent, что говорит о том, что Вы сможете легко обменивать цветовые настройки между деревьями простым методом Assign. Все цвета VT доступны для изменения в поле Colors.

- **BorderColor** - цвет дополнительной границы, устанавливаемой в свойстве **BorderWidth**.
- **DisabledColor** - цвет текста отключённого (vsDisabled) узла.
- **DropMarkColor** - цвет небольшой отметки, появляющейся при Drag&Drop переносе узла по верху или по низу другого.
- **DropTargetBorder** - цвет границы прямоугольника, закрашивающего узел, в который производится Drag&Drop вставка (DropTargetNode).
- **DragTargetColor** - цвет самого прямоугольника, закрашивающего узел, в который производится Drag&Drop вставка (DropTargetNode).
- **FocusedSelectionBorderColor** - цвет границы выделенного узла.
- **FocusedSelectionColor** - цвет выделенного узла.
- **GridLineColor** - цвет линий сетки у дерева.
- **HeaderHotColor** - цвет текста заголовка колонки, над которой сейчас находится курсор.
- **HotColor** - цвет узла, над которым находится курсор.
- **SelectionRectangleBlendColor** - цвет фона, заполняющего прямоугольник выделения.
- **SelectionRectangleBorderColor** - цвет границы прямоугольника выделения.
- **TreeLineColor** - цвет соединительных линий узлов.
- **UnfocusedSelectionBorderColor** - цвет границы выделенного, но не имеющего фокус узла.
- **UnfocusedSelectionColor** - цвет выделенного, но не имеющего фокус узла.

Особенно интересны пары параметров, где один отвечает за границу, а другой за фон. Комбинируя различные цвета между ними, можно получать довольно интересные сочетания.

8.2. Отображение подсказки.

VT позволяет выводить подсказку тремя различными вариантами. Режим подсказки задаётся в параметре **HintMode**:

- **hmDefault** - передаёт вывод подсказки стандартным обработчикам VCL. В этом случае в подсказке будет выведен текст из поля **Hint**.
- **hmHint** - использует для получения текста подсказки событие **OnGetHint**. Предусмотрен для вывода индивидуальной подсказки для каждого узла.
- **hmHintAndDefault** - комбинация двух предыдущих параметров. Если курсор будет находится над узлом, то действие будет эквивалентно действию параметру hmHint, если же курсор будет находится непосредственно над пустой областью дерева, то действие будет эквивалентно параметру hmDefault.
- **hmTooltip** - особый параметр. Применяется, если Вы хотите отобразить текстовую подсказку над узлами, текст которых не умещается в колонке и обрезается. В этом случае обработка события OnGetHint не нужна, VT сам всё сделает за Вас. Учтите, что если Вы всё же обработаете это событие, то своим действием оно перекроет этот параметр.

Наиболее часто используемый параметр - hmTooltip. Он не обязует Вас лишними обработчиками и в то же время облегчает навигацию по дереву с узлами с длинным текстом. Если же Вы хотите выводить собственный текст в подсказке, то можете воспользоваться параметром hmHint. В этом случае обработчик получения подсказки будет примерно такой:

```
procedure TfrmMain.VTGetHint(Sender: TBaseVirtualTree; Node: PVirtualNode;
```

```
  Column: TColumnIndex; var LineBreakStyle: TVTTooltipLineBreakStyle;
  var HintText: WideString);
begin
  if VT.HintMode <> hmTooltip then
    HintText := 'Хинт из OnGetHint события: текст подсказки...'
  else
    HintText := VT.Text[Node, Column];
end;
```

Параметр hmDefault не требует никаких обработчиков. Это индивидуальная подсказка дерева из его поля Hint, как и у любых других VCL классов.

VT позволяет использовать анимацию для всплывающей подсказки. Её можно настроить параметром **HintAnimation** дерева.

- **hatNone** - выключить анимацию подсказки.
- **hatFade** - эффект затухания, плавное появление подсказки.
- **hatSlide** - скользящий эффект появления подсказки.
- **hatSystemDefault** - используется эффект анимации, установленный в системе. Рекомендуемый параметр.

8.3. Отображение выделения.

Цвет выделения задаётся в полях параметра Colors, который уже был рассмотрен выше. Кроме этих параметров, VT позволяет указать прозрачность для выделенных узлов. Это означает, что через выделенные узлы будет просвечиваться фон дерева. Степень прозрачности задаётся в поле **SelectionBlendFactor**. Сама эта возможность по умолчанию отключена в настройках дерева. Включить её можно, изменив опцию TreeOptions.PaintOptions.**toUseBlendedSelection** в True.

Помимо этого, VT позволяет изменить форму выделения. Стандартный прямоугольник выделения можно закруглить с помощью поля **SelectionCurveRadius**.

Обратите внимание: VT на данный момент не поддерживает закругление выделения при включённых настройках прозрачности выделения. Т.е. действие поля SelectionCurveRadius опускается при включённой опции toUseBlendedSelection.

Прямоугольник выделения (тот, что растягивается мышью) тоже может менять своё отображение. С помощью параметра **DrawSelectionMode** Вы можете изменить его стиль. **smDottedRectangle** для прямоугольника с границами в виде точек и отсутствием заливки (используется стандартными TTreeView и TListView), **smBlendedRectangle** для прямоугольника выделения в стиле Windows XP (прозрачный синий прямоугольник по умолчанию). Свойство активно только при выключенных темах оформления. Это означает, что если дерево поддерживает визуальные стили, то значение параметра всегда будет smBlendedRectangle.

8.4. Параметры анимации.

Несколько параметров анимации и прокрутки.

- **AnimationDuration** - длительность анимации при раскрытии/скрытии узла. Анимация включается опцией TreeOptions.AnimationOptions.**toAnimatedToggle** дерева.
- **AutoExpandDelay** - длительность паузы перед автоматическом раскрытии узла. Включается опцией TreeOptions.AutoOptions.**toAutoExpand** дерева.
- **AutoScrollDelay** - длительность паузы перед автоматической прокруткой дерева. Включается опцией TreeOptions.AutoOptions.**toAutoScroll** дерева.
- **AutoScrollInterval** - количество узлов, прокручиваемое при перемещении колёсика мыши.

8.5. Параметры отображения линий сетки и соединительных линий узлов.

Даже для такой мелочи как соединительные линии и линии сетки предусмотрены свои настройки. Цвет, как Вы уже знаете, изменяется параметрами GridLineColor и TreeLineColor. Параметры, рассмотренные здесь, позволяют управлять стилем отрисовки этих самих линий.

Для линий, соединяющих узлы, имеется параметр интересный **LineMode**. Он интересен тем, что помимо обычного стиля отрисовки **lmNormal** (а-ля TTreeView/TListView) позволяет указать

стиль **lmBands**. Тогда соединительные линии узлов будут образовывать ряды и колонки, похожие на таблицу или диаграмму. Смотрится довольно необычно по сравнению с обычным стилем.

Для линий сетки существует параметр **LineStyle**:

- **lsCustomStyle** – определяет, что Вы сами будете задавать стиль линий сетки через событие **OnGetLineStyle**.
- **lsDotted** – линии сетки в виде пунктира.
- **lsSolid** – стиль, используемый TTreeView/TListView (обычные сплошные линии).

Если вам не хватает стилей lsDotted и lsSolid, то Вы вольны указать свой собственный стиль в обработчике события OnGetLineStyle. Его параметру Bits необходимо передать битмап 6*6 пикселей, представленный указателем на массив байт (TByteArray). Вот простой пример, увеличивающий расстояние между пунктирами на максимальное:

```
procedure TfrmMain.VTGetLineStyle(Sender: TBaseVirtualTree; var Bits:
Pointer);
var
  i: Integer;
begin
  Bits := @Arr;
  for i := 0 to Length(Arr) - 1 do
    Arr[i] := 254;
end;
```

8.6. Отображение узла.

Нами остались нерассмотренными ещё несколько свойств, влияющих на положение узла в ячейке.

NodeAlignment позволяет указать относительно чего будет смещаться узел. Смещение, как Вы помните, задаётся в поле Align TVirtualNode записи.

- **naFromBottom** – смещение относительно нижнего края ячейки.
- **naFromTop** – смещение относительного верхнего края.
- **naProportional** – пропорциональное смещение, не направленное ни в ту, ни в другую сторону. В этом случае на положение узла будет влиять только значение поля Align.

Параметр **ButtonStyle** позволяет изменить стиль кнопки развёртывания/свёртывания узла. У него может быть всего два значения:

- **bsRectangle** – кнопка в виде прямоугольника с плюсом/минусом.
- **bsTriangle** – кнопка в виде небольшого чёрного треугольника. Похожий имеют списки и деревья браузера Opera.

Параметр активен только при отключённых стилях оформления.

И, наконец, последний параметр – **Indent**. Он позволяет указать длину отступа для каждого уровня вложенности узлов. Чем больше значение этого параметра, тем дальше друг от друга будут расположены дочерние и родительские узлы.

8.7. Изменение отрисовки заголовков.

Во всех наших примерах отрисовки мы не изменили отображение одного очень важного элемента дерева – его заголовка. Думаю, об этом стоит сказать пару слов.

За отрисовку заголовков отвечают три события: **OnAdvancedHeaderDraw**, **OnHeaderDrawQueryElements** и **OnHeaderDraw**. В рамках данной статьи Я рассмотрел первые два, так как они более сложны в исполнении, нежели последнее. С ним уж как-нибудь сами справитесь :).

VT реализует очень удобный механизм отрисовки заголовков, позволяя какие-то отдельные элементы рисовать пользователю, а какие-то оставлять на попечение VT. Например, Вы можете изменить отрисовку фона заголовка колонки, но при этом оставить неизменным её текст. Или же вообще изменить отрисовку только какого-то одного заголовка, оставив остальные в стандартном

виде. Что примечательно, изменённая отрисовка доступна как при включённых стилях оформления, так и при выключенных.

Событие OnHeaderDrawQueryElements как раз предназначено для того, чтобы передать дереву список изменяемых вами элементов заголовка.

- **hpeBackground** - фон заголовка.
- **hpeDropMark** - иконка Drag&Drop перетаскивания.
- **hpeHeaderGlyph** - иконка слева от текста заголовка.
- **hpeSortGlyph** - иконка сортировки.
- **hpeText** - текст заголовка.

В нашем примере мы заменим фон и текст заголовка, оставив всё остальное без изменений:

```
//----------------------------------------------------------------------------
// Передаём в набор те элементы, которые мы будем заменять.
// Непереданные элементы (hpeDropMark, hpeHeaderGlyph, hpeSortGlyph)
// останутся на совести дерева.
//----------------------------------------------------------------------------
procedure TfrmMain.VTHeaderDrawQueryElements(Sender: TVTHeader;
  var PaintInfo: THeaderPaintInfo; var Elements: THeaderPaintElements);
begin
  // Мы будем заменять текст колонок и фон всего заголовка
  Elements := [hpeBackground, hpeText];
end;
```

А вот так будет выглядеть сама процедура отрисовки:

```
procedure TfrmMain.VTAdvancedHeaderDraw(Sender: TVTHeader;
  var PaintInfo: THeaderPaintInfo; const Elements: THeaderPaintElements);
begin
  if hpeBackground in Elements then
    with PaintInfo.TargetCanvas do
    begin
      // Рисуем фон всего заголовка дерева
      Brush.Color := $00C9C9C9;
      FillRect(PaintInfo.PaintRectangle);
      // Растягиваем градиент
      StretchDraw(PaintInfo.PaintRectangle, ImgGrad.Graphic);
      if PaintInfo.Column <> nil then
      begin
        // Необходимо нарисовать заголовок какой-то конкретной колонки
        case PaintInfo.Column.Index of
          // Пользователь
          0: Draw(PaintInfo.PaintRectangle.Left +
((PaintInfo.PaintRectangle.Right - PaintInfo.PaintRectangle.Left) div 2) -
(ImgUser.Width div 2),
            PaintInfo.PaintRectangle.Top, ImgUser.Graphic);
          // Кол-во сообщений
          1: Draw(PaintInfo.PaintRectangle.Left +
((PaintInfo.PaintRectangle.Right - PaintInfo.PaintRectangle.Left) div 2) -
(ImgMsg.Width div 2),
            PaintInfo.PaintRectangle.Top, ImgMsg.Graphic);
        end;
      end;
    end;
end;
```

Не забудьте включить изменённую отрисовку опцией заголовка hoOwnerDraw.

8.8. Многострочные надписи в узлах.

VT позволяет своим узлам иметь многострочные надписи, разрешая это опцией vsMultiline для узла. В этом случае редактор дерева будет заменён на многострочный с удобной вертикальной полосой прокрутки.

Кроме своего явного применения (хранение нескольких строк в одном узле), это свойство можно использовать как альтернативу обрезающемуся тексту, который не умещается в границах колонки. Вы можете просто переносить неумещающийся текст на новые и новые линии. Для включения многострочности принято использовать событие инициализации узла:

```
procedure TfrmMain.VTInitNode(Sender: TBaseVirtualTree; ParentNode,
  Node: PVirtualNode; var InitialStates: TVirtualNodeInitStates);
begin
  // Включаем/выключаем поддержку многострочности
  if CbMultiline.Checked then
    Include(InitialStates, ivsMultiline)
  else begin
    Exclude(InitialStates, ivsMultiline);
    // НЕ ЗАБЫВАЙТЕ ВЫПОЛНЯТЬ ЭТО:
    Node.States := Node.States - [vsMultiline];
    // Это выключит многострочность для узлов, где она раньше была.
  end;
end;
```

Но это ещё не всё. Этот код поможет разбить текст на линии, но высоту узлов он не изменит. Т.е. может случиться так, что перенесённые линии текста уже не будут видны. Для корректного подбора высоты с учётом многострочности VT имеет очень полезный метод - **ComputeNodeHeight**. Он подсчитывает высоту, необходимую для полного отображения узла в ячейке. Применять мы его будет соответственно в обработчике события OnMeasureItem, там, где и указывается высота узлов:

```
//----------------------------------------------------------------------------
// Подсчёт высоты узлов в соответствии с высотой переносимого текста.
//----------------------------------------------------------------------------
procedure TfrmMain.VTMeasureItem(Sender: TBaseVirtualTree;
  TargetCanvas: TCanvas; Node: PVirtualNode; var NodeHeight: Integer);
begin
  if CbMultiline.Checked then
  begin
    NodeHeight := VT.ComputeNodeHeight(TargetCanvas, Node, 0) + 4;
    NodeHeight := Max(18, NodeHeight);
  end
  else
    NodeHeight := 18;
end;
```

Теперь всё сделано верно.

8.9. Фон дерева.

Последнее наше упущение - параметры **Background**, **BackgroundOffsetX** и **BackgroundOffsetY**.

Используются они, когда необходимо нарисовать на фоне дерева что-то наподобие тематической картинки, «водяного» (прозрачного) знака. Яркий пример - папки картинок или музыки в Windows XP, где в нижнем правом углу изображена тематическая картинка для каждой из папок, отражающая её суть.

Background - сама картинка, которая будет отображаться.

BackgroundOffsetX и BackgroundOffsetY - её смещения по X и Y координатам соответственно.

Для включения отображения картинки необходимо установить опцию дерева TreeOptions.PaintOptions.**toShowBackground** в True. Опция toStaticBackround из того же раздела позволяет разрешить или запретить заполнение дерева изображением на всю видимую область.

В событии OnResize очень удобно изменять положение картинки, например, центрировать её:

```
//---------------------------------------------------------------------------
// Располагаем фоновую картинку по центру.
//---------------------------------------------------------------------------
procedure TfrmMain.VTResize(Sender: TObject);
begin
  VT.BackgroundOffsetX := (VT.Width div 2) - (VT.Background.Width div 2);
  VT.BackgroundOffsetY := (VT.Height div 2) - (VT.Background.Height div 2);
end;
```

Проект Figure 1.8, демонстрирующий использование различных свойств и событий VT, находится в папке Fig 1.8.

9.0. Настройки VT.

Как Я и обещал, привожу полный список настроек VT (TreeOptions) с описанием назначения каждого параметра.

- **AnimationOptions**:
  - **toAnimatedToggle** – отвечает за анимацию свёртывания/развёртывания.
- **AutoOptions**:
  - **toAutoDropExpand** – развёртывает узел, если он будет оставаться целью drag&drop (DropTargetNode) некоторое время, заданное параметром AutoExpandDelay. Рекомендуется установить в True.
  - **toAutoExpand** – автоматически раскрывает/сворачивает узел при получении им фокуса.
  - **toAutoScroll** – прокручивает дерево, если происходит операция drag&drop и курсор мыши находится рядом с краями дерева.
  - **toAutoScrollOnExpand** – прокручивает дерево на максимально возможное число позиций при развёртывании узла.
  - **toAutoSort** – автоматически сортирует дерево при изменении параметров Header.SortDirection, или Header.SortColumn, или при добавлении нового узла.
  - **toAutoSpanColumns** – довольно необычная опция. Она позволяет перенести текст, не помещающийся в данной колонке на соседнюю, если она пуста (не содержит текста).
  - **toAutoTristateTracking** – автоматическое отслеживание grayed-состояния для узлов с типом отметки ctTriStateCheckBox.
  - **toAutoHideButtons** – кнопки развёртывания/свёртывания будут автоматически скрываться для данного узла, если все его дочерние будут спрятаны (vsVisible).
  - **toAutoDeleteMovedNodes** – источники перемещённых узлов будут автоматически удаляться после операций drag&drop.
  - **toDisableAutoscrollOnFocus** – выключает автоматическую прокрутку колонки к видимой области, если она получает фокус.
  - **toAutoChangeScale** – изменяет высоту узлов в соответствии с настройками размера шрифтов Windows.
  - **toAutoFreeOnCollapse** – при сворачивании узла все его дочерние узлы будут удалены, но опция vsHasChildren для данного узла сохранится.
  - **toDisableAutoscrollOnEdit** – не прокручивает дерево в горизонтальном направлении при редактировании узла, чтобы расположить видимую область по центру.
- **MiscOptions**:
  - **toAcceptOLEDrop** – зарегестрировать дерево, как возможную цель для OLE drag&drop.
  - **toCheckSupport** – включить поддержку отметок для узлов.
  - **toEditable** – включить режим редактирования для узлов. В этом случае станет доступен стандартный редактор VT и Ваши собственные.
  - **toFullRepaintOnResize** – полностью перерисовывать дерево при любом изменении его размеров.
  - **toGridExtensions** – включить поддержку некоторых расширений для симуляции элемента управления сетки (неподобие TDBGrid). В частности, изменяется реакция на некоторые горячие клавиши. Удобно использовать при создании приложений баз данных с VT.
  - **toInitOnSave** – производить инициализацию узлов при сохранении в поток или файл.
  - **toReportMode** – дерево ведёт себя как TListView с поддержкой report mode.
  - **toToggleOnDblClick** – узлы будут сворачиваться/разворачиваться при двойном клике на них.
  - **toWheelPanning** – разрешает навигацию по дереву с помощью движений мыши (panning). Это особый режим прокрутки дерева, в который оно входит по нажатию средней кнопки мыши. После этого курсор начнёт принимать вид стрелок и дерево можно будет прокручивать по определённым направлениям, передвигая курсор к соответствующим краям дерева. Чем ближе пододвигать курсор, тем быстрее будет происходить прокрутка.
  - **toReadOnly** – запрещается любое изменение дерева, в том числе взаимодействие с узлами и их редактирование.
  - **toVariableNodeHeight** – включается, когда требуется отображение узлов различной высоты. В этом случае высоту узлов дерево будет получать из обработчика события OnMeasureItem.

- **toFullRowDrag** – разрешает начало перетаскивания узла при нажатии мышью в любую область узла, а не обязательно по тексту или картинке. Должно использоваться вместе с опцией toDisableDrawSelection.

- **PaintOptions**:
  - **toHideFocusRect** – не отображает пунктирный прямоугольник фокуса по границам узла.
  - **toHideSelection** – не отображает бежевый (по умолчанию) прямоугольник выделения для выделенных узлов, когда само дерево не имеет фокуса. В этом случае, когда дерево не будет иметь фокуса, Вы не сможете разобрать выделенные узлы от не выделенных.
  - **toHotTrack** – включает отслеживание узла, находящегося под курсором. По умолчанию, такой узел будет отображаться подчёркнутым, если Вы включите эту опцию.
  - **toPopupMode** – отображает дерево, как будто бы оно всегда имеет фокус.
  - **toShowBackground** – отрисовывать фоновую картинку, если есть (параметр Background).
  - **toShowButtons** – Отображать кнопки развёртывания/свёртывания напротив узлов.
  - **toShowDropmark** – показывать значок, отображающий положение будущего узла при операциях drag&drop.
  - **toShowHorzGridLines** – включить отображение горизонтальных линий сетки.
  - **toShowRoot** – учитывать отступ и рисовать соединительные линии для самых верхних узлов первого уровня вложенности (дочерних узлов RootNode).
  - **toShowTreeLines** – отображать соединительные линии для узлов.
  - **toShowVertGridLines** – включить отображение вертикальных линий сетки.
  - **toThemeAware** – включает отображении всех элементов управления дерева (кнопок, отметок и т.д.) в соответствии с текущей темой оформления Windows XP. Приложение должно поддерживать визуальные темы оформления.
  - **toUseBlendedImages** – включить прозрачность для ghosted-узлов и для узлов, участвующих на данный момент в копировании/вырезании.
  - **toGhostedIfUnfocused** – картинки будут отображаться прозрачными до тех пор, пока узел не будет выделен.
  - **toFullVertGridLines** – продливает вертикальные линии сетки до конца дерева. Если данная опция будет отключена, то линии закончатся на последнем видимом узле.
  - **toAlwaysHideSelection** – отображает узлы как невыделенные независимо от их состояния.
  - **toUseBlendedSelection** – отображать выделенные узлы прозрачными.
  - **toStaticBackground** – запрещает заполнять фоновой картинкой видимую область дерева.
- **SelectionOptions (только для TVirtualStringTree дерева)**:
  - **toDisableDrawSelection** – запрещает пользователю добавлять в текущее выделение узлы с помощью прямоугольника выделения.
  - **toExtendedFocus** – позволяет выделять ячейки и редактировать текст во всех колонках, а не только в MainColumn.
  - **toFullRowSelect** – дерево будет рисовать прямоугольник выделения на всю свою ширину для данного узла. Таким образом, длина прямоугольника выделения не будет зависеть от ширины текста. Помимо этого, узлы можно будет выбирать нажатием мыши в любую область дерева, а не только по тексту узла.
  - **toLevelSelectConstraint** – запрещает выделение узлов различных уровней. Это означает, что в текущее выделение смогут попасть только узлы с таким же уровнем вложенности, что и у первого выделенного.
  - **toMiddleClickSelect** – разрешает выделение узлов средней кнопкой мыши.
  - **toMultiSelect** – разрешает выделение более чем одного узла.
  - **toRightClickSelect** – разрешает выделение узлов правой кнопкой мыши.
  - **toSiblingSelectConstraint** – ограничивает текущее выделение только узлами одного и того же родителя.
  - **toCenterScrollIntoView** – располагает видимую (клиентскую) область дерева центрально по вертикали по отношению к узлу, получающему фокус. Очень удобно для длинных списков.
  - **toSimpleDrawSelection** – упрощает операцию выделения узлов, в частности, не требует, чтобы прямоугольник выделения пересёкся с текстом узла для его выделения. Если опция будет установлена в False, то для выделения узла придётся тащить прямоугольник выделения до текста MainColumn колонки.
- **StringOptions**:

- **toSaveCaptions** – сохраняет текст из ячеек узла при операциях с буфером обмена, drag&drop и при сохранении в поток или файл. Рекомендуется включать при drag&drop и выключать при работе с файлами, т.к. сохранение данных узла в потоке строго рекомендуется делать самостоятельно.
- **toShowStaticText** – включает отображение статического текста, который отображается рядом с обычным, но в отличие от него не может быть отредактирован редактором VT.
- **toAutoAcceptEditChange** – автоматически принимать отредактированный текст, если редактирование не было отклонено по Esc или завершено по Enter. В этом случае новый текст передаётся в обработчик события OnNewText.

9.1. Горячие клавиши, определённые в VT.

VT содержит в себе множество предопределённых горячих клавиш почти для всех своих действий. Нижеприведённый список горячих клавиш взят из оффициального хелпа, и по сему здесь Вы видите его перевод. Не поленитесь ознакомиться с этим списком, он довольно интересный :).

Клавиша	Модификаторы	Результат
Home	нет	Выделяет первый видимый узел в активной колонке. Этот узел также получает фокус ввода. **Добавления в grid mode:** Активный узел не изменяется, но фокус передаётся первой видимой колонке.
	shift	Выделяет первый видимый узел в активной колонке и включает в выделение все видимые узлы от предыдущего активного до нового активного. Эффект очень схож с Проводником Windows. **Добавления в grid mode:** Активный узел не изменяется, но фокус передаётся первой видимой колонке. Выделение не изменяется.
	control	Прокручивает дерево до левого верхнего края, не изменяя ни фокус, ни выделение.
End	нет	Выделяет последний видимый узел в активной колонке. Этот узел также получает фокус ввода. **Добавления в grid mode:** Активный узел не изменяется, но фокус передаётся последней видимой колонке.
	shift	Выделяет последний видимый узел в активной колонке и включает в выделение все видимые узлы от предыдущего активного до нового активного узла. Эффект очень схож с Проводником Windows. **Добавления в grid mode:** Активный узел не изменяется, но фокус передаётся первой видимой колонке. Выделение не изменяется.
	control	Прокручивает дерево до правого нижнего края, не изменяя ни фокус, ни выделение.
Prior (страница вверх)	нет	Прокручивает дерево на страницу вверх, выделяя при этом узел сверху. Этот узел также получает фокус.
	shift	То же самое, но включает в выделение страницу прокрученных узлов.
	control	То же самое, но при прокрутке активный узел не теряет своего фокуса и выделение не изменяется.
Next (страница вниз)	нет	Тот же эффект, что и **prior**, но прокручивается страница вниз.
	shift	Тот же эффект, что и **prior**, но прокручивается страница вниз.
	control	Тот же эффект, что и **prior**, но прокручивается страница вниз.
Up	нет	Перемещает фокус с текущего узла на предыдущий.
	shift	Перемещает фокус с текущего узла на предыдущий добавляет его в выделение.
	control	Прокручивает дереву на одну линию вверх. Высота одной линии определена значением параметра DefaultNodeHeight.

Down	нет	Тот же эффект, что и **up,** но по направлению вниз.
	shift	Тот же эффект, что и **up,** но по направлению вниз.
	control	Тот же эффект, что и **up,** но по направлению вниз.
Left	нет	Перемещает фокус в родительский узел текущего узла и выделяет его, если у текущего узла нет дочерних узлов, и он уже свёрнут. В противном случае фокус не изменится, но узел будет свёрнут. В обоих случаях узел, имеющий фокус, останется единственным выделенным узлом дерева. **Добавления в grid mode:** если расширенный фокус включён (опция toExtendedFocus в настройках), то результатом будет простой переход на предыдущую видимую колонку.
	shift	В противоположность случаю с отсутствием клавиши-модификатора состояние раскрытия узла не изменяется. Фокус передаётся предыдущему узлу в любом случае, но при этом предыдущий узел добавляется в текущее выделение.
	control	Дерево прокручивается влево на количество пикселей, которые берутся из параметра Indent.
Right	нет	Перемещает фокус на первый дочерний узел текущего узла, если у данного узла есть дочерние узлы, и он уже развёрнут. В противном случае, фокус останется неизменным, но узел будёт развёрнут. В обоих случаях узел, имеющий фокус, останется единственным выделенным узлом дерева. **Добавления в grid mode:** если расширенный фокус включён (опция toExtendedFocus в настройках), то результатом будет простой переход на следующую видимую колонку.
	shift	Аналогично случаю с отсутствием клавиши-модификатора, но выделение расширяется дочерним узлом.
	control	Тот же эффект, что и у **left,** но дерево прокручивается вправо.
Back	нет	Перемещает выделение в родительский узел текущего узла и делает его единственным выделенным узлом дерева.
	shift	Эффект отсутствует.
	control	Эффект отсутствует.
Tabulator (табулятор)	нет	Клавиша табуляции имеет несколько особое назначение, потому что используется только в дереве с grid extensions для перехода от ячейки к ячейке. Без модификатора фокус изменяется слева-направо и сверху-вниз. Чтобы клавиша была активна, необходимо включить поддержку табуляции, выставив параметр WantTabs в True.
	shift	То же, что и без клавиши-модификатора, но фокус перемещается в обратную сторону, справа-налево и снизу-вверх.
	control	Эффект отсутствует.
F1	нет	Эта функциональная клавиша активирует индивидуальную справку для узла. Через событие OnGetHelpContext приложение запрашивает необходимое значение контекста помощи для отображения.
	shift	Эффект отсутствует.
	control	Эффект отсутствует.
F2	нет	Эта функциональная клавиша включает режим редактирования для узла, имеющего на данный момент фокус, если такой узел есть, дерево редактируемо, и приложение разрешает редактирование через событие OnEditing.

	shift	Эффект отсутствует.
	control	Эффект отсутствует.
+ (add)	нет	Разворачивает текущий узел.
	shift	Этот модификатор в одиночку не имеет действия. См. комментарий ниже.
	control	Нажатие клавиши control вместе с клавишей «+» приводит к автоматическому изменению размеров всех колонок дерева. Если также будет нажата клавиша shift, то будут развёрнуты абсолютно все узлы дерева.
- (subtract)	нет	Сворачивает текущий узел.
	shift	Этот модификатор в одиночку не имеет действия. См. комментарий ниже.
	control	Нажатие клавиши control вместе с клавишей «-» приводит к восстановлению размеров колонок дерева. Если также будет нажата клавиша shift, то будут свёрнуты абсолютно все узлы дерева.
* (умножить)	нет	Рекурсивно разворачивает текущий узел со всеми его дочерними узлами.
	shift	Эффект отсутствует.
	control	Эффект отсутствует.
/ (разделить)	нет	Рекурсивно сворачивает текущий узел со всеми его дочерними узлами.
	shift	Эффект отсутствует.
	control	Эффект отсутствует.
Escape (отмена)	нет	Останавливает действия дерева, требующие специального состояния, например, редактирование, выделение мышью, drag&drop и т.д.
	shift	Эффект отсутствует.
	control	Эффект отсутствует.
Space (пробел)	нет	Используется, если поддержка отметок включена (опция toCheckSupport в настройках) и текущий узел имеет какой-нибудь тип отметки, отличный от ctNone. В этом случае клавиша переключает состояние отметки узла.
	shift	Эффект отсутствует.
	control	Эффект отсутствует.
Apps (клавиша меню)	нет	По аналогии с **F1** активирует popup меню клавишей приложения для каждого узла. Popup меню присваивается узлу через событие OnGetPopupMenu.
	shift	Эффект отсутствует.
	control	Эффект отсутствует.
A	нет	Это единственная буквенная клавиша, используемая деревом как горячая. Она имеет эффект только при использовании с модификатором control.
	shift	Эффект отсутствует.
	control	Нажатие «A» вместе с клавишей control приводит к выделению всех видимых узлов дерева.

10.0. Взаимодействие с базами данных.

Несмотря на то, что VT не ориентирован на использование в приложениях баз данных, его заманчивые возможности могут найти применение и там. VT легко может быть использован в небольших приложениях баз данных с помощью нехитрых телодвижений. Причиной его использования вполне может быть скорость работы и богатые возможности в реализации интерфейса (редакторы, отметки, функции отрисовки и т.д.).

10.1. Простейшее БД приложение с VT.

Здесь Я приведу простой пример использования VT в приложении с БД, который будет демонстрировать самые основные функции: чтение, запись и сортировка записей табицы. Суть примера заключается в извлечении значений некторых полей всех записей базы данных и выводе этих значений в VT. Частичный код примера дан ниже со всеми необходимыми комментариями. Обратите внимание, что пример использует стандартную базу данных Delphi из «DBDEMOS» набора (employee.db).

```
type
  PDBRec = ^TDBRec;

  TDBRec = record
    ANum: Integer;
    AFirstName,
    ALastName: WideString;
    Initialized: Boolean; // True, если узел был заполнен начальными данными
 end;

 //---------------------------------------------------------------------------
 // Подготавливает дерево и начинает заполнение.
 //---------------------------------------------------------------------------
 procedure TfrmMain.LoadDB;
 begin
   if not Table.Active then
     Table.Open;
   with Table do
   begin
     Filtered := False;
     Filter := '';
     First;
   end;
   with VT do
   begin
     Clear;
     NodeDataSize := SizeOf(TDBRec);
     RootNodeCount := Table.RecordCount;
     Header.SortColumn := 0;
     // Для сортировки дерева рекомендуется передать последний параметр как
     // True, чтобы каждый отсортированный узел обновил свои данные.
     SortTree(0, sdAscending, True);
   end;
 end;

 //---------------------------------------------------------------------------
 // Чистим память от строк.
 //---------------------------------------------------------------------------
 procedure TfrmMain.VTFreeNode(Sender: TBaseVirtualTree; Node: PVirtualNode);
 var
   DBRec: PDBRec;
 begin
   DBRec := Sender.GetNodeData(Node);
   if Assigned(DBRec) then
     Finalize(DBRec^);
 end;
```

```
//---------------------------------------------------------------------------

procedure TfrmMain.VTGetText(Sender: TBaseVirtualTree; Node: PVirtualNode;
  Column: TColumnIndex; TextType: TVSTTextType; var CellText: WideString);
var
  DBRec: PDBRec;
begin
  DBRec := Sender.GetNodeData(Node);
  case Column of
    0: CellText := IntToStr(DBRec.ANum);
    1: CellText := DBRec.AFirstName;
    2: CellText := DBRec.ALastName;
  end;
end;

//---------------------------------------------------------------------------

procedure TfrmMain.VTHeaderClick(Sender: TVTHeader; Column: TColumnIndex;
  Button: TMouseButton; Shift: TShiftState; X, Y: Integer);
begin
  if Button = mbLeft then
  begin
    if Sender.SortColumn <> Column then
      Sender.SortColumn := Column;
    if Sender.SortDirection = sdAscending then
      Sender.SortDirection := sdDescending
    else
      Sender.SortDirection := sdAscending;
    VT.SortTree(Sender.SortColumn, Sender.SortDirection, True);
  end;
end;

//---------------------------------------------------------------------------
// Основная процедура в этом примере. Её задача - заполнять каждый новый
// узел данными из таблицы. После считывания очередной записи из таблицы,
// мы перемещаемся на следующую методом Next, пока не закончится база
// данных.
// В сортировке этот метод необходим для обновления данных. Хотя, на самом
// деле, можно обойтись и без инициализации во время сортировки.
//---------------------------------------------------------------------------
procedure TfrmMain.VTInitNode(Sender: TBaseVirtualTree; ParentNode,
  Node: PVirtualNode; var InitialStates: TVirtualNodeInitStates);
var
  DBRec: PDBRec;
begin
  DBRec := Sender.GetNodeData(Node);
  if not DBRec.Initialized then
  begin
    Initialize(DBRec^);
    with DBRec^ do
    begin
      ANum := StrToInt(Table.FieldByName('EmpNo').AsString);
      AFirstName := Table.FieldByName('FirstName').AsString;
      ALastName := Table.FieldByName('LastName').AsString;
      Initialized := True;
    end;
  end
  else begin
    // Запись уже была заполнена начальными данными при загрузке значений
    // таблицы. Значит, нужно просто обновить данные узла. После
    // редактирования,
    // к примеру.
```

```
    with Table do
    begin
      Filtered := True;
      Filter := 'EmpNo = ' + #39 + IntToStr(FUpdatingNum) + #39;
      if RecordCount > 0 then
        with DBRec^ do
        begin
          ANum := StrToInt(Table.FieldByName('EmpNo').AsString);
          AFirstName := Table.FieldByName('FirstName').AsString;
          ALastName := Table.FieldByName('LastName').AsString;
        end;
    end;
  end;
  if not Table.Eof then
    Table.Next;
end;

//----------------------------------------------------------------------------
// Вносит изменения в базу данных.
//----------------------------------------------------------------------------
procedure TfrmMain.VTNewText(Sender: TBaseVirtualTree; Node: PVirtualNode;
  Column: TColumnIndex; NewText: WideString);
var
  DBRec: PDBRec;
begin
  if NewText = '' then
    Exit;
  DBRec := Sender.GetNodeData(Node);
  with Table do
  begin
    Filtered := True;
    // Включает фильтрацию и по индексу сотрудника находит необходимую
    // запись в таблице.
    Filter := 'EmpNo = ' + #39 + IntToStr(DBRec.ANum) + #39;
    if RecordCount > 0 then
    begin
      Table.Edit;
      try
        // Если запись найдена, то можно внести изменения
        case Column of
          0: Table.FieldByName('EmpNo').AsInteger := StrToIntDef(NewText,
            0);
          1: Table.FieldByName('FirstName').AsString := NewText;
          2: Table.FieldByName('LastName').AsString := NewText;
        end;
        Table.Post;
        // Эта переменная нужна будет в событии инициализации узла.
        // С помошью неё будет выбрана изменившаяся запись путём фильтрации
        // и обновлены данные узла.
        // Необходимость в этой переменной состоит в том, что
        // в событии инициализации нельзя использовать фильтрацию по
        // DBRec.ANum, т.к. пользователь мог отредактировать первую колонку
        // и
        // DBRec.ANum указывал бы на старый индекс записи, тогда как новый
        // содержался бы в NewText.
        if Column = 0 then
          FUpdatingNum := StrToIntDef(NewText, 0)
        else
          FUpdatingNum := StrToIntDef(VT.Text[Node, 0], 0);
        Sender.ReinitNode(Node, False);
      except
        // Ошибка...
```

```
      end;
    end
    else
      MessageBox(Self.Handle, 'Ошибка: Запись не найдена.',
        PChar(Application.Title), MB_OK or MB_ICONERROR or MB_APPLMODAL or
          MB_DEFBUTTON1);
  end;
end;
```

Не забудьте выставить опцию дерева **toGridExtensions** в True, чтобы оно больше походило на редактор ячеек базы данных (TDBGrid).

Полная версия проекта Figure 1.9, демонстрирующего взаимодействие приложения баз данных с VT, находится в папке Fig 1.9.

11.0. Заключение.

Вот, наконец, мы и подошли к концу. Теперь Вы можете быть твёрдо уверены, что знаете о VT более чем достаточно. Я старался сделать статью как можно более полной и информативной, не упуская ни единого параметра и события, в отличие от официального хелпа по компоненту, но, к сожалению, везде есть свои рамки, и поэтому что-то осталось за кадром, а о чём-то Я просто мог забыть упомянуть. Если бы Я уделял пристальное внимание абсолютно каждому элементу VT, то и без того получившаяся большая статья растянулась бы страниц на сто и походила бы скорее на полноценное печатное издание :). Нами остались нерассмотренными всего несколько событий компонента, не имеющие принципиального значения. Это, к примеру, OnResetNode, OnShortenString, OnGetPopupMenu. Уверен, Вам самим будет интересно разобраться в них самостоятельно.

Я надеюсь, что Вам понравился компонент, и Вы убедились в его исключительной мощности и удобстве. Думаю, теперь VT станет вашим повседневным компонентом номер один, а про TTreeView и TListView Вы забудете, как о страшном сне. И, конечно же, Я надеюсь, что Я выполнил поставленную перед собой цель и Вам понравилась моя статья :).

Примечание:

Заранее прошу прощения за ошибки и опечатки в статье. Я старался исправлять орфографию и грамматику как мог :). Обо всех ошибках и неточностях в статье просьба сообщать на мой почтовый адрес (4quadr0@gmail.com).

С уважением, Титов Сергей (Quadr0).