USENIX Association

Proceedings of the XFree86 Technical Conference

Oakland, California, USA
November 8–9, 2001

A New Tree Widget for GTK+ 2.0

Jonathan Blandford
jrb@redhat.com

Abstract

The GtkTreeView widget is a new widget for GTK+ 2.0. Designed in a Model/View controller fashion, it is highly flexible and capable. It allows multiple columns and custom rendering of cells, as well as specialized models.

1 Introduction

The *GtkTreeView* is a new widget for GTK+ 2.0. Written over the course of the past year, it is meant to replace the other main Tree/List widgets currently in GTK+, the GtkCList/GtkCTree.[1] It is a much more substantial work than either of those widgets and is similar in scope to both Java Swing's JTree and ETable from Evolution. It is designed to solve a number of problems found in GtkCList and GtkCTree. Most importantly:

Lack of flexibility in the data model: The older widget required the data to be part of the widget making it difficult to keep the data synchronized.

Lack of custom cell renderers: Although possible to customize renderers by drawing pixmaps, it requires a lot of effort and is basically a hack.

Wrong order of inheritance: A list is a tree without any children. Having the tree inherit from a list is backward.

In addition, the GtkTreeView was designed to keep one of the main strengths of the GtkCList/GtkCTree—the ability to efficiently display large amounts of data.

[1] There also exists a GtkList and a GtkTree. These two widgets are extremely broken and have been deprecated for some time. They will not be mentioned again in this paper.

2 Overview

The tree object-set[2] is designed around a Model/View/Controller (MVC) design and consists of four major parts:

- the tree view widget
- the view columns
- the cell renderers
- the model interface

The first three parts comprise the View, while the last part is the Model. One of the prime benefits of the MVC design is that multiple views can be created of a single model. For example, consider a file manager. It could create one model mapping the file system such that only one copy of all the file data needs be kept in memory at one time. A different view of various parts of this model can be created when a new window is needed.

2.1 A quick example

It is somewhat tricky to understand how these parts all fit together. Let us first start off with a very simple example. The relevant code is located in Appendix A.

[2] There are actually nine GObjects, five interfaces, and one widget that make up the whole tree as shipped with GTK+, spread out among over thirty-five source files and comprises of almost thirty thousand lines of code. For simplicity's sake, I will refer to the widget throughout this paper as the *tree object-set*, unless referring to a specific element.

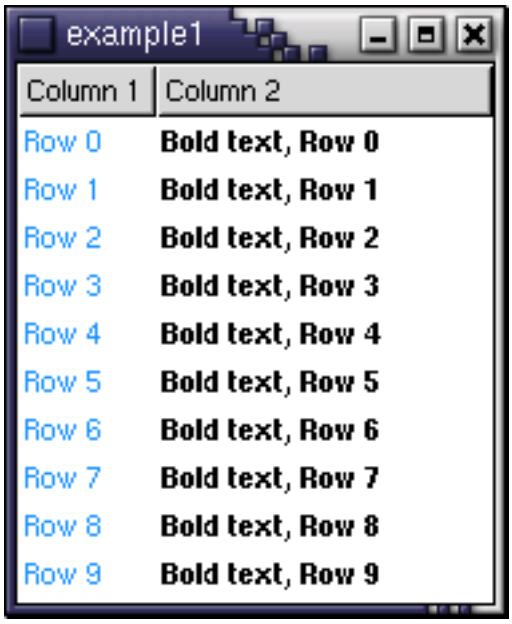

Figure 1: Simple example of the GtkTreeView widget

In this example, we will simply create a model, create a view, and then put them together. Please note that this example only shows the part of the code relevant to the tree-object set and does not include the rest of the code needed for a functioning program. When run, it could look something like Figure 1.

Here, we create a model (in this case, a GtkTreeStore) and then add data. We add a column and then add a text renderer to the column. We finally create a mapping from the model to the cell renderers.

3 The Model

One of the primary parts of the tree object-set is the *GtkTreeModel* interface. This interface allows the view to get data from the model and is meant to be attached to any appropriate data structure. The programmer simply has to implement this interface on his own data type for it to be viewable by the GtkTreeView. The interface can be seen in Appendix B.

The model is represented as a hierarchical tree of strongly-typed, columned data. In other words, the model can be seen as a tree where every node has different values depending on which column is being queried. The type of data found in a column is determined by using the GType system (ie. G_TYPE_INT, GTK_TYPE_BUTTON, G_TYPE_POINTER, etc.) and should be familiar to GTK+ programmers.[3] The types are homogeneous per column across all nodes. It is important to note that this interface only provides a way of examining a model and observing changes. The implementation of each individual model decides how and if changes are made.

In order to make life simpler for programmers who do not need to write their own specialized model, two generic models are provided with GTK+ 2.0—the *GtkTreeStore* and the *GtkListStore*. These two models are 'shove' models. To be more specific, the developer simply pushes data into these models as necessary. They provide the data structure as well as all appropriate tree interfaces. As a result, implementing drag and drop, sorting, and storing data is trivial. For the vast majority of trees and lists, these two models should be sufficient.

3.1 Accessing the Model

Models are accessed on a node/column level of granularity. One can query for the value of a model at a certain node and a certain column on that node. There are two structures used to reference a particular node in a model. They are the *GtkTreePath* and the *GtkTreeIter*.[4] The majority of the interface consists of operations on a GtkTreeIter.

A path is essentially a potential node. It is a hypothetical location on a model that may or may not actually correspond to a node on a specific model. The GtkTreePath struct can be converted into either an array of unsigned integers or a string. The string form is a list of numbers separated by a colon. Each number refers to the offset at that level. Thus, the path "0" refers to the root node and the path "2:4" refers to the fifth child of the third node.

[3] for those unfamiliar with it, more information can be found at http://developer.gnome.org/doc/API/

[4] here "iter" is short for "iterator"

By contrast, a GtkTreeIter is a reference to a specific node on a specific model. It is a generic struct with an integer and three generic pointers. These are filled in by the model in a model-specific way. One can convert a path to an iterator by calling $gtk\_tree\_model\_get\_iter()$. These iterators are the primary way of accessing a model and are similar to the iterators used by GtkTextBuffer. They are generally statically allocated on the heap and only used for a short time. The model interface defines a set of operations using them for navigating the model.

It is expected that models fill in the iterator with private data. For example, the GtkListStore model, which is internally a simple linked list, stores a list node in one of the pointers. The GtkTreeModelSort stores an array and an offset in two of the pointers. Additionally, there is an integer field. This field is generally filled with a unique stamp per model. This stamp is for catching errors resulting from using invalid iterators with a model.

The lifecycle of an iterator is a little confusing. Iterators are expected to be valid for as long as the model is unchanged. The model is considered to own all outstanding iterators and nothing needs to be done to free them from the user's point of view. Additionally, some models guarantee that an iterator is valid for as long as the node it refers to is valid (most notably the GtkTreeStore and GtkListStore). Although generally uninteresting, as one always has to allow for the case where iterators do not persist beyond a signal, some very important performance enhancements were made in the sort model. As a result, a flag was added to indicate this behavior.

3.2 Rationale of complexity

Unfortunately, this dual way of referring to nodes increases the complexity for the application developer. There is a temptation to want to simply refer to nodes only by an iterator or by a path.[5] However, it quickly became clear that having both methods of accessing a node was necessary for performance reasons.

[5]In fact JTree uses a non-opaque list of Objects to implement their TreePath. I feel this quickly becomes non-trivial for a lot of models.

The view must be referenced by paths[6] for reasons that will be made clear later in this paper. Likewise, it was important to have a way of quickly finding a node in the model.

For example, during every expose event, the GtkTreeView finds the first exposed node and then walks through the remainder of the exposed nodes in order. For a model like the GtkListStore (which is internally implemented as a linked list), this operation would be $O(n^2)$ if each node was referenced by index. By using the GtkTreeIter, and, by extension, having a pointer to the internal list node, the $gtk\_tree\_model\_iter\_next()$ operation can be very fast.

3.3 The GtkTreeRowReference structure

To make life easier for application developers, a structure called the *GtkTreeRowReference* exists. It is a GtkTreePath that listens to a model and adjusts for changes allowing you to track a row while other rows are inserted/deleted. Internally, it is used extensively for things like keyboard navigation and selection.

4 Cell Renderers

The cell renderers are a class of objects that inherit from *GtkCellRenderer*. They do the actual rendering of a cell to the display. They are flyweight objects, meaning that they do not store any state but have it passed in just before being used. As a result, one cell renderer can be used for every row in a tree.

GTK+ 2.0 comes with three basic cell renderers—GtkCellRendererText, GtkCellRendererPixbuf, and GtkCellRendererToggle. Custom renderers are easy to create as well. It is important to note that there is nothing specific about these renderers to the GtkTreeView/GtkTreeModel. In fact, there are plans to write a table-like widget for future versions of GTK+ that would use them.

[6]A number of the methods on the GtkTreeView actually take an iterator as their argument. However, these iterators are immediately converted internally to paths.

The primary way of setting data on a cell is through the GTK+ property system.[7] GObjects have a way of associating string attributes with typed values. For example, the text cell renderer has “text”, “font”, “foreground”, “background”, and “weight” properties each taking a string as their argument. It also has an “underline” property that takes a boolean.[8] One can set these properties in the following way:

```
g_object_set (text_renderer,
              "text", "Random text",
              "font", "Sans 12",
              "foreground", "black",
              "weight", "bold",
              "underline", TRUE,
              NULL);
```

Once set, you can query the cell renderer for its size and render it to a GdkDrawable. Additionally, renderers have two other interesting methods. A cell renderer may be activated or edited.[9] For example, the GtkCellRendererToggle is activated whenever it is clicked. Once activated, it emits a signal indicating what has happened. As there is no way to modify a model using the GtkTreeModel interface, it is up to the programmer to listen for these signals and to modify the model directly.

Editing is done in a similar fashion. When the cell renderer starts the editing process, it creates a GtkWidget. The GtkTreeView displays this widget over the area the cell occupies. This widget will exist until editing has ended, at which point it emits an “editing_finished” signal. Any widget implementing the GtkCellEditable interface can be used. For example, the text renderer takes a GtkEntry and uses it to display edited text.

The rationale for doing the editing in a separate widget is that the state needs to persist throughout the duration of the editing phase.[10] A cell renderer may be correctly initialized with data at the start of the editing phase but may be used elsewhere in the model. It is necessary to store some states such as pointer grabs and cursor positions.

5 Tree Columns

The *GtkTreeViewColumn* objects bind the entire object-set together. Each of these objects is a visible column in the view (as opposed to the model). These objects can be packed into the view in a manner similar to widgets in an hbox. Once added, they can be moved around, hidden, or removed.

The primary task of the GtkTreeViewColumn is to handle drawing each column for the view. Additionally, it determines the sizing policy of the column as well as handling the button in the header (if the headers are shown.) It also handles the sort arrows.

The way that the GtkTreeViewColumn draws the cells is through cell renderers. Each column has one or more cell renderer packed into it. For example, here is some code—this time written in Python[11]—showing how to create a column with a look similar to a check box:

```
column = gtk.TreeViewColumn ()
toggle_cell =
    gtk.CellRendererToggle ()
text_cell =
    gtk.CellRendererText ()
toggle_cell.connect ("activated",
                     callback)

column.set_title ("Example Column")
column.pack_start (toggle_cell,
                   gtk.FALSE)
column.pack_start (text_cell,
                   gtk.TRUE)
```

[7] formerly the GtkArg system

[8] indeed, this object has thirty-four properties of various types and usefulness. The most commonly used ones are shown in the example.

[9] Please note that the editable interface has not been finalized at the time of the writing of this paper and may even be removed.

[10] the fact that a lot of code could be reused did not hurt either

[11] I am doing this example in Python to give an idea of what it looks like in another language. However, at the time of this paper, the Python bindings for GTK+ 2.0 are not frozen. As a result, working code may look different in the future.

```
column.set_spacing (4)
column.set_attributes (toggle_cell,
                       "toggled",
                       0)
column.set_attributes (text_cell,
                       "text", 1,
                       "foreground",
                       2)
```

In this example, we create a column and the renderers. We connect the "activated" signal so we can modify the column when the toggle button is pressed. We then pack the two cells into the column with a four pixel space between them. The boolean value passed to $pack\_start()$ indicates whether or not the cell is allocated leftover space. In this instance, the text following the toggle button gets all remaining space. We then make the mapping of attributes on the cells to various columns in the model.

As an alternative to providing a mapping for every renderer, we can use a custom function to set the attributes on the renderer. This is most useful if we want to store a struct in the GtkTreeModel.

6 The GtkTreeView

The *GtkTreeView* widget is the only actual widget in the entire tree object-set and is the part that the user sees. It was designed to be as efficient as possible in rendering at a specific pixel-offset as well as handling inserting and deleting nodes. In order to handle fast rendering, it implements a height cache that internally maps each visible node in the tree to a specific pixel position (and vice versa).

6.1 The height and offset cache

The height cache is implemented as a red/black tree[12] of red/black trees in order to keep insertion and deletion efficient.

[12]For those unfamiliar with this data type, a red/black tree is essentially a continuously balanced binary tree. The tree is rebalanced (if needed) after every insertion and deletion. It is never any deeper than $2\ \log(n)$ nodes, and, as a result, operations such as insertion, search, and deletion are guaranteed to never be worse than $O(\log(n))$. There is a somewhat larger memory overhead needed as a result of this structure. However, this overhead is in general much less than the cost of the data stored.

This cache is used to store a mapping of heights on the tree to paths. Thus, when the tree gets an expose event on a certain section of the tree, it can determine the corresponding row and then have the GtkTreeViewColumns render that area. The actual implementation of this cache is extremely involved and outside the scope of this paper.

6.2 Selection

The GtkTreeView widget exports a separate object to handle selection of rows. This object is conceptually part of the view widget—it has been separated mostly for code and API cleanliness reasons. The tree uses the cache to store the selected state of a row. As we are already storing a node for every row in the tree, we can simply add a flag indicating whether a particular row is selected. When the developer needs to find all selected rows, the tree view walks through all of the cached rows finding those marked as selected.

One of the unfortunate side effects is that the code does not guard well against reentrancy issues. We hope to fix this in a future version of GTK+.

7 Sorting and Drag and Drop

There are three other interfaces that a GtkTreeModel can export. They provide support for sorting and drag and drop.

7.1 Sorting

Sorting is done at the model level using the *GtkTreeSortable* interface. This interface defines the way that the model is sorted and exists primarily to allow GtkTreeViewColumns to display sort-order arrows when needed. Both GtkTreeStore and GtkListStore implement the interface allowing them to be easily be sorted. However, if there are multiple views of a model, then each view will be sorted identically. This is often not the desired behavior. To sort the views in different ways, a proxy model such as GtkTreeModelSort must be used.

The GtkTreeModelSort is a GtkTreeModel that wraps around other GtkTreeModels storing a reordering of that model. Very simply, when the view requests a value from a row, the GtkTreeModelSort converts this request to the appropriate row on the child model and proxies the request allowing us to sort a model without actually duplicating the stored values.

Internally, the GtkTreeModelSort model stores a mapping of its paths to its children's paths. That means that whenever it gets a request for an iterator, it must convert a child path to a child iterator and look up the value. This process is potentially very slow (especially in the case of the GtkTreeStore and GtkListStore models). To work around this problem, there is a cache of iterators in the model. Not all models support iterator caching (as perviously mentioned, there is a flag to indicate whether or not it is supported). In practice, those models that don't allow you to cache iterators (such as those based on an array) tend to be fast in converting from path to iterator anyway meaning that the sort model works well in most cases.

7.2 Drag and Drop

Just like sorting, drag and drop is done through interfaces on a GtkTreeModel. There are two such interfaces—*GtkTreeDragSource* and *GtkTreeDragDest*. The former is necessary to drag from a tree, while the latter is needed to support drops. Like the sorting column, all the models shipped with GTK+ implement these two signals.

8 "If I had to do it over…" or "lessons learned"

It turned out to be harder than I had expected to write a generic model interface. As a result of not restricting the view to a specific implementation of the model (unlike the text widget in GTK+ 2.0), there was no way to work around particular limitations of that implementation. These limitations led to some unfortunate hacks like the ITERS_PERSIST flag and the GtkTreePath/GtkTreeIter dichotomy. It also made binding the model in a high level language like Python tricky.

Additionally, the cell measuring system actually turned out to be noticeably slow. While the view can handle large (on the order of 100,000) numbers of nodes without problems, measuring these strings is a serious bottleneck. A better geometry-management scheme might have saved a few of the necessary hacks added for speed.

A First Example

A small code-fragment that creates all the parts necessary for a view.

```
enum {
  FIRST_COLUMN,
  SECOND_COLUMN,
  NUM_COLUMNS
};
...
{
  GtkTreeStore *model;
  GtkWidget *view;
  GtkTreeViewColumn *column;
  GtkCellRenderer *cell_renderer;

  /* Create a model.  We are using the store model for now, though we
   * could use any other GtkTreeModel */
  model = gtk_tree_store_new (NUM_COLUMNS, G_TYPE_STRING, G_TYPE_STRING);

  /* Fill the model with text */
  custom_populate_model_function (model);

  /* Create a view */
  view = gtk_tree_view_new_with_model (GTK_TREE_MODEL (model));

  /* Create a cell render and set an attribute */
  cell_renderer = gtk_cell_renderer_text_new ();
  g_object_set (G_OBJECT (cell_renderer),
                "foreground", "dodger blue",
                NULL);

  /* Create a column, associating the "text" attribute of the
   * cell_renderer to the first column of the model */
  column = gtk_tree_view_column_new_with_attributes ("Column 1",
                                                     cell_renderer,
                                                     "text", FIRST_COLUMN,
                                                     NULL);

  /* Create a second cell render and set an attribute */
  cell_renderer = gtk_cell_renderer_text_new ();
  g_object_set (G_OBJECT (cell_renderer),
                "weight", "bold",
                NULL);,

  /* Create another column, associating the "text" attribute of the
   * cell_renderer to the second column of the model */
  column = gtk_tree_view_column_new_with_attributes ("Column 2",
```

```
                                                    cell_renderer,
                                                    "text", SECOND_COLUMN,
                                                    NULL);

  /* Add the second column to the view. */
  gtk_tree_view_append_column (GTK_TREE_VIEW (view), column);

  /* The view now holds a reference to the model.  We can get rid of our own
   * reference if we no longer need it */
  g_object_unref (G_OBJECT (model));

  /* Now we can pack the view in another container */
}
```

B GtkTreeModel Interface

The GtkTreeModel interface describes the way that the view and the model interact. It is shown here in an abbreviated fashion in the interest of space. The fully documented interface with prototypes can be found at http://developer.gnome.org/doc/api/.

B.1 The Signals

These signals are emitted every time the model changes. It is possible to explicitly follow what happens with a model simply by listening to these signals.

GtkTreeModel::row_changed
GtkTreeModel::row_inserted
GtkTreeModel::row_has_child_toggled
GtkTreeModel::row_deleted
GtkTreeModel::rows_reordered

B.2 The Methods

These methods are used to inspect a model. Every model must implement these at a minimum.

```
GtkTreeModelFlags gtk_tree_model_get_flags       (GtkTreeModel *tree_model);
gint              gtk_tree_model_get_n_columns   (GtkTreeModel *tree_model);
GType             gtk_tree_model_get_column_type (GtkTreeModel *tree_model,
                                                  gint          index);
void              gtk_tree_model_get_value       (GtkTreeModel *tree_model,
                                                  GtkTreeIter  *iter,
                                                  gint          column,
                                                  GValue       *value);

/* Iterator movement */
gboolean          gtk_tree_model_get_iter        (GtkTreeModel *tree_model,
                                                  GtkTreeIter  *iter,
```

```
                                                GtkTreePath  *path);
GtkTreePath *     gtk_tree_model_get_path       (GtkTreeModel *tree_model,
                                                 GtkTreeIter  *iter);
gboolean          gtk_tree_model_iter_next      (GtkTreeModel *tree_model,
                                                 GtkTreeIter  *iter);
gboolean          gtk_tree_model_iter_children  (GtkTreeModel *tree_model,
                                                 GtkTreeIter  *iter,
                                                 GtkTreeIter  *parent);
gboolean          gtk_tree_model_iter_has_child (GtkTreeModel *tree_model,
                                                 GtkTreeIter  *iter);
gint              gtk_tree_model_iter_n_children (GtkTreeModel *tree_model,
                                                 GtkTreeIter  *iter);
gboolean          gtk_tree_model_iter_nth_child (GtkTreeModel *tree_model,
                                                 GtkTreeIter  *iter,
                                                 GtkTreeIter  *parent,
                                                 gint          n);
gboolean          gtk_tree_model_iter_parent    (GtkTreeModel *tree_model,
                                                 GtkTreeIter  *iter,
                                                 GtkTreeIter  *child);
void              gtk_tree_model_ref_node       (GtkTreeModel *tree_model,
                                                 GtkTreeIter  *iter);
void              gtk_tree_model_unref_node     (GtkTreeModel *tree_model,
                                                 GtkTreeIter  *iter);
```