Duncan Temple Lang

Note

To run the second example, use the R command

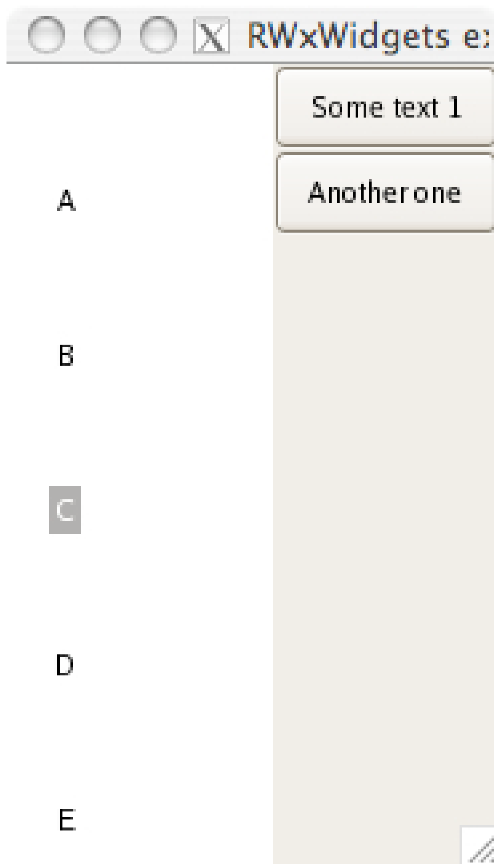
```
xmlSource("DragNDrop.xml", xpath = "//r:example[@id='DynamicDnD']")
```

The Basics of Drag N Drop in RwxWidgets

In this article, we describe what we learn about Drag N Drop in wxWidgets and how we start to implement this in *RwxWidgets*. This also exemplifies how we can implement and extend C++ class with R functions as methods. Specifically, we provide a function to implement the method OnDropText for the C++ class RwxTextDropTarget.

We'll start with what is hopefully a very simple setup. We have three widgets, a wxListCtrl and two buttons. We want to allow a user to drag from the list to the first of the two buttons. The result of the drag will be set the label on the second button to the text "Some text", pasted together with the number of times this has been done, i.e. an incremented count. It is an unexciting example in an effort to keep it as simple as possible.

Figure 1. A wxListCtrl and two buttons and the user drags an entry in the list onto the top button to change its label.



We create a top-level frame and a size to hold the two widgets.

```
f =  RFrame("RWxWidgets example",
              as.integer(c(50, 30)),
              as.integer(200, 100))

sizer = wxBoxSizer(wxHORIZONTAL)
```

We create the wxListCtrl widget and add 5 entries to it.

```
listCtrl = wxListCtrl(f, size = c(100, 300))
sapply(1:5, function(i) listCtrl$InsertItem(i, LETTERS[i]))
sizer$Add(listCtrl, .5, wxEXPAND)
```

Next, we create the first button and give it a simple callback. Then we add a second button which will serve as a check that it does not accept the drag. We arrange the two buttons into a vertical sizer on the right of the window. We also add a wxPanel with a wxStaticBoxSizer to identify it. The wxPanel will act as a location from which we can drag values to our button. It will illustrate how we can programmatically make a widget a source of draggable content rather than relying on the widgets in the toolkit which support dragging.

```
vbox = wxBoxSizer(wxVERTICAL)
 b = wxButton(f, wxID_ANY, "Second Button")
 b$AddCallback(wxEVT_COMMAND_BUTTON_CLICKED,
                function(ev)  cat("Button 2\n"))
 vbox$Add(b, .3, wxEXPAND)

 other = wxButton(f, wxID_ANY, "Another one")
 vbox$Add(other, .3, wxEXPAND)


 panel = p = wxPanel(f, size =c(100, 400))
 box = wxStaticBoxSizer(wxVERTICAL, p, "Frame")


 p$SetSizer(box); box$SetSizeHints(p)
 paintFrame = function(event) {
    .Call("R_paintFrame", frame, event)
 }
 p$AddCallback(wxEVT_LEFT_DOWN, function(ev) cat("Down\n"))

 vbox$Add(p, 1, wxEXPAND)
 sizer$Add(vbox, .5, wxEXPAND)
```

Now, we setup a wxDropTarget object which we want to associate with this second button. We use a wxTextDropTarget but we implement the OnDropText method ourselves in R. Therefore, we create an instance of the RwxTextDropTarget class in C++. And we provide an R function for the OnDropText method. This is the function **drop** in the code below. It is called with the x and y coordinates of the mouse giving the point of the drop and also the actual material being dropped. It creates and sets the new label for this button and importantly, returns **TRUE**.

```
count = 0
 drop = function(x, y, text) {
```

```
   cat("In drop\n")
   b$SetLabel(paste(text, count))
   count <<- count + 1
   TRUE
 }
```

Given this function which will act as the C++ method implementation for OnDropText, we can create the wxDropTarget object and associate it with the button, **b**.

```
target = RwxTextDropTarget(OnDropText = drop)
 b$SetDropTarget(target)
```

Since we use wxTextDropTarget, we don't need to explicitly call SetDataObject on the target. This is done implicitly in the Text and DropTarget combination.

```
drag =
function(event)
{
  print(event$GetEventObject())
  data =  wxTextDataObject("Some text") # btn1$GetLabel()),
  ans = wxDropSource(data, listCtrl, TRUE)
  cat("After DoDragDrop", ans, "\n")
}
listCtrl$AddCallback(wxEVT_COMMAND_LIST_BEGIN_DRAG, drag)
```

And now we can display the window and its contents.

```
f$SetSizer(sizer)
 sizer$SetSizeHints(f)
 f$Show()
print(f$GetSize())
```

Unfortunately, the "other" button is not draggable. That is, we cannot drag it onto the first button. We would like to be able to programmatically specify this as opposed to relying on using "draggable" widgets. So, we now get the opportunity to attempt to make the button draggable. We'll leave this for the moment, as dealing with such events in natively implemented widgets is a little tricky.

Instead, we have introduced a wxPanel below the two buttons. We can initiate a drag from that by catching the mouse down events. (We can do slightly better than this later on.) In the callback for this event, we create a wxTextDataObject containing some text "from the frame". Then we create a wxDropSource object. And then we call its DoDragDrop method. This is done implicitly from within the creation of the wxDropSource and controlled by the **TRUE** argument.

```
drag =
function(event)
{
  wxDropSource(wxTextDataObject("from the frame"), panel, TRUE)
}

panel$AddCallback(wxEVT_LEFT_DOWN, drag)
```

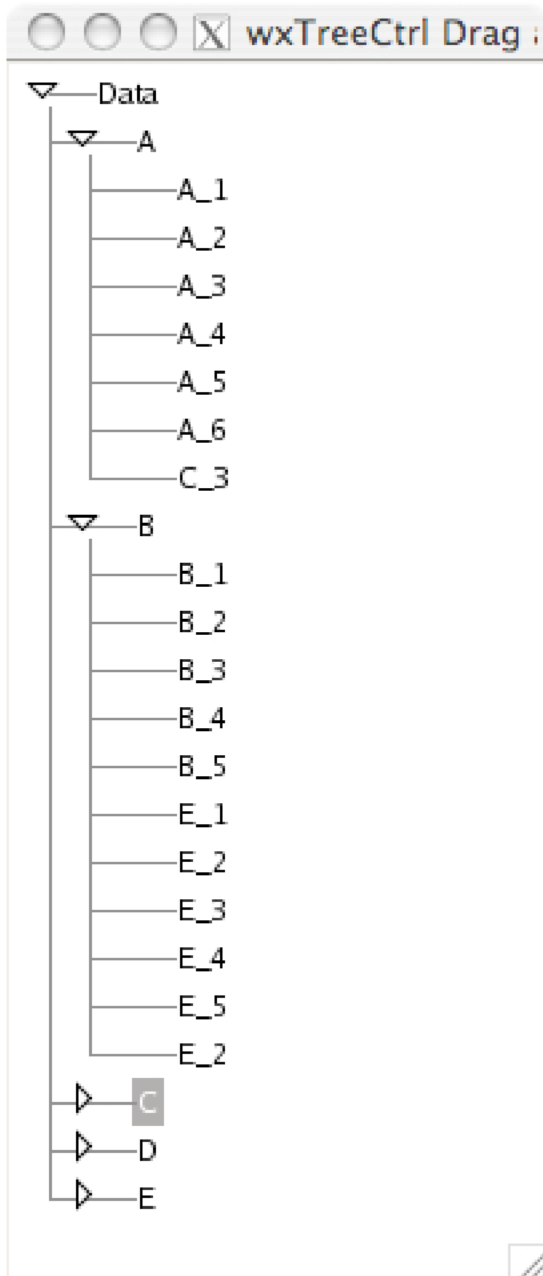
And for the moment, we will explicitly run a blocking event loop. This will not be necessary in the future as the event loop will be merged with the regular R event loop.

```
eloop = wxEventLoop()
wxEventLoop_Run(eloop)
```

Dynamic Drag N Drop in RwxWidgets

In this example, we will work with a more complex version of drag and drop where the content to be dragged is an R object and where it can be dropped depends on the run-time characteristics of the R object associated with each drop site. The setup is this: we have a wxTreeCtrl whose nodes correspond to data sets in R. We can drag one onto another but only if they have the same number of observations. If they are compatible in this way, the variables from the data set being dragged are added to the data set on which the drag source is dropped. We further allow the individual variables to be dragged and dropped between data sets, rather than the entire dataset. The resulting GUI looks something like

Figure 2.



We start by creating the tree control.

```
f = RFrame("wxTreeCtrl Drag and Drop example", size = c(300, 500))
tree = wxTreeCtrl(f, wxID_ANY)
top = tree$AddRoot("Data")
```

Now, we add some data sets. There are several ways to go about this drag and drop setup. The simplest manner is to keep the computations within R and to use the existing drag and drop formats provided by wxWidgets. One way to do this is to use names of R variables as the data to drag and drop and then in the OnDrop method accept this name and find the corresponding data set and merge it. Alternatively, we can

use the wxWidgets mechanism for defining our own data format and exchange the actual R data. If we want a general mechanism for exchanging R data with arbitrary widgets/applications, we would want to do this and support converting the data to different forms such as text, HTML, etc. In this example, we will use the simple approach as this is all within R and within a single "application"/GUI. This uses names to identify data objects and so is using "non-local" variables which is not good programming practice. But in this case, it simplifies matters a good bit and the interchange between R and C++ code.

The basic strategy is that we will keep an environment in which we have the three data sets stored by the name used in the display on the wxTreeCtrl. These could be global variables in the R globalenv, but we keep them separately. Then when we drag, we drag text which is the name of the R variable bound to the appropriate data set. And when we drop that text, we use its name to fetch the values and do the work.

We create the entries in the tree via a call to *sapply*.

```
env = new.env()
n = c(20, 30)
sizes = c(A = n[1], B = n[2], C = n[1], D = n[1], E = n[2])
els = sapply(names(sizes),
             function(id) {
                 item = tree$AppendItem(top, id)
          numVars = rpois(1, 5) + 1  # make certain at least 1.
              data = as.data.frame(replicate(numVars, rnorm(sizes[id])))
   names(data) = paste(id, 1:numVars, sep = "_")
   assign(id, data, env)

                    # add entries in the wxTreeCtrl for these.
   sapply(names(data),
                           function(name)
                               tree$AppendItem(item, name)
                           )

                 tree$Expand(item)
                 item

             })
```

When I initially wrote this, I added the drop target object with its own method for each of the data sets, e.g.

```
tree$SetDropTarget(RwxTextDropTarget(OnDropText = drop))
```

where drop was suitably defined in a data set-specific manner. And the result was that we were always adding to the last entry in the node, i.e. to data set E. The reason is that this is being added to the tree and not the tree item and so as we set the drop target, it remove the previous one and accordingly the last function was the one in effect when the drop was performed. (Many of the computations would be easier if we were able to deal with the items in the wxTreeCtrl as regular objects.) So we need to do this at the tree-level and so cannot use closures to capture the name of the data set corresponding to a particular tree item. Instead, we must compute the name of the data set by determining on which item the drag or the drop is occurring. We can do this by using the (x, y) coordinates to determine the item. We use HitTest to get the wxTreeItemId associated with the (x, y) coordinates and also the flags value indicating the nature of the match of the (x, y) coordinates to the entries in the wxTreeCtrl. We check that there was indeed a "hit" onto an item and return if not. We will do this when dropping the object and also as we move over potential drop target items. So we create a separate function - *checkDropTarget* - for checking whether the (x, y) location corresponds

to a suitable wxTreeItemId. For the drop and the mouse-over actions, we want to find the wxTreeItemId for the given (x, y) coordinates and check that there was a hit. We get the labe for that wxTreeItemId and check it is a valid variable within **`env`**. We also want to make certain that the And if we can determine the number of rows in both the target and the "drop" data sets, then we can give feedback about whether the two are compatible. We do this by assigning the name of the data set/variable being dragged in **`env`** to the variable **`.source`** when the drag is initiated. (We make certain to remove this when the drag ends!) The use of **`env`** is a convenient way to share data across the drag callback function and the the OnDragOver and OnDropText methods we supply for the RwxTextDropObject class instance.

```
checkDropTarget =
function(x, y, source = NA)
{
      # find the item we are on.
   item = tree$HitTest(c(x, y))

      # Check that we are on an wxTreeCtrl item
   hit = bitAnd(item$flags, c(wxTREE_HITTEST_ONITEM, wxTREE_HITTEST_ONITEMINDENT))
   if(all(hit == 0)) {
      cat("not on any item in the wxTreeCtrl\n")
      return(NULL)
   }

   item = item$.result

   if(!item$IsOk())
     return(NULL)

     # Get the name of the data set corresponding to the item.
   id = tree$GetItemText(item)

   if(!exists(id, env)) {
      return(NULL)
   }

   if(is.na(source))
     if(exists(".source", env))
       source = env$.source

   if(is.na(source))
      return(item)

   oldData = get(id, env)
   newData = get(source, env)

   if(nrow(oldData) != nrow(newData)) {
     cat("different numbers of observations in", source, "and", id, "\n")
     return(NULL)
   }
```

```
   return(item)
}
```

The following R function is used as the OnDragOver handler for our DropTarget and is responsible for determining whether the current (x, y) location is a compatible drop spot.

```
OnEnter =
function(x, y, def)
{
   if(is.null(checkDropTarget(x, y)))
      return(wxDragNone)

   return(def)
}
```

And finally, this is the function that will process the dropping of the data. It checks the drop target and, if compatible, merges the data into the data associated with this drop spot and then adds new entries to the tree item.

```
# create the OnDrop function that will handle a drop on this item.
drop =
function(x, y, text) {
   item = checkDropTarget(x, y, text)
   if(is.null(item))
      return(FALSE)

   id = tree$GetItemText(item)

   oldData = get(id, env)
   newData = getDroppedData(text, env)

   assign(id, cbind(oldData, newData), env)

     # add new items to this node in the tree for the the newly added variables.

   sapply(names(newData),
           function(name) {
               tree$AppendItem(item, name)
           })
   TRUE
}
```

You might note that in the function *drop* above, we call *getDroppedData* to fetch the data that was being dragged. This gives us a little more abstraction about how manage the data being transferred and allows us to implement alternative approaches without changing *drop* very much or at all. The way we implement for our example is to fetch the data set stored in the **env** environment using the name given to us from the item being dragged.

```
getDroppedData =
function(text, env)
{
   get(text, env)
```

```
}
```

At this point, we have the two methods that we use to create our own customized version of the RwxTextDropTarget class.

```
tree$SetDropTarget(RwxTextDropTarget(OnDropText = drop,
                                     OnDragOver = OnEnter))
```

The next thing we have to do is arrange for the drag. All we do is permit the drag operation (via the call to Allow) and Note that we use wxEVT_COMMAND_TREE_BEGIN_DRAG rather than the one in the documentation wxEVT_TREE_BEGIN_DRAG. We need the COMMAND! Note also that we call the Skip method for the event so that the usual processing occurs in addition to ours. If we do not do this, the widget acts as if we never let go of the mouse button and will continue to select things as we move around the widget, or even outside of the widget!

```
tree$AddCallback(wxEVT_COMMAND_TREE_BEGIN_DRAG,
                   function(ev) {
                       ev$Allow()
                       id = tree$GetItemText(ev$GetItem())

         if(!exists(id, env)) {
                    ev$Skip()
           return(FALSE)
                       }

         cat("Dragging from", id, "\n")
         env$.source = id
         on.exit(remove(".source", envir = env))
                       status = wxDropSource(wxTextDataObject(id), tree, TRUE)
         cat("after DoDragDrop", status, "\n")
         ev$Skip()

                       TRUE
                   })
```

We can use this approach to handle the dragging of individual variables and not entire datasets. In the initiation of the drag operation (i.e. the callback function above), we can find the item being dragged and determine whether it is a data set or a variable within a data set. To do this, we might examine the label to get the variable name and then search through each data set in **env** to find the data set. This is not a good way to do this generally, as we might have two data sets with the same variable name and then we would have difficulties. If we ask for the parent wxTreeItemId of the wxTreeItemId in which the drag was initiated, we can ask for its label and get the name of the data set in which the variable is located. We are assuming the names of the data sets are unique (since we are assining the data frames to the name within **env**), so this will work in general. Alternatively, we could avoid using names altogether and use an item-specific data object to hold the R data frame for a particular wxTreeItemId. But this makes things slightly more complex at this point. If we are dealing with a variable, then we assign the name of the variable to **.varName** in **env** and then that is available when we drop the dragged data.

```
tree$AddCallback(wxEVT_COMMAND_TREE_BEGIN_DRAG,
                   function(ev) {
```

```
                       ev$Allow()
                       id = tree$GetItemText(ev$GetItem())

           # Figure out if we are dealing with a data set
                         # or a variable.
         parent = tree$GetItemParent(ev$GetItem())

         ctr = 0
         parent = ev$GetItem()
         while(parent$IsOk() && !is.null(parent)) {
             parent = tree$GetItemParent(parent)
      ctr = ctr + 1
                       }
         if(ctr == 3) {
                           tmp = tree$GetItemParent(ev$GetItem())
      varName = id
      id = tree$GetItemText(tmp)
                       } else
             varName = NA

         if(!exists(id, env)) {
                    ev$Skip()
           return(FALSE)
                       }

         cat("Dragging", varName, "from", id, "\n")
         env$.source = id
         env$.varName = varName
         on.exit(remove(".varName", ".source", envir = env))
                       status = wxDropSource(wxTextDataObject(id), tree, TRUE)
         cat("after DoDragDrop", status, "\n")
         ev$Skip()

                       TRUE
                   })
```

We can not take advantage of the way we implemented the `drop` function and redefine `getDroppedData` so that we get not just the data set, but the specific variable if **`.varName`** is set to something meaningful.

```
getDroppedData =
function(text, env)
{
   data = get(env$.source, env)
   if(exists(".varName", env) && !is.na(env$.varName)) {
       data[env$.varName]
   } else
       data
}
```

At this point, we have the ability to drag entire data sets or individual variables. We could also allow the drag to move the data, i.e. copy it and discard from the original data set. And we could allow for the selection of multiple variables.

```
tree$Expand(top)
f$Show()

eloop = wxEventLoop()
wxEventLoop_Run(eloop)
```