Subject: Object Tree Posted by J.D. Smith on Mon, 16 Nov 1998 08:00:00 GMT View Forum Message <> Reply to Message

I've made an Object-Based tree that might be useful... Basically, to use it, derive a class from it, complete with node data, and possible data collection, and/or modification methods. Each instance of this class is a single node in a Tree, and contains two pointers: children and siblings, specifying a list of this node's children (may have none), and siblings (at least one... itself).

Things you can do with it:

- *Add children and/or siblings to a given node.
- *Delete a given node and all descendents.
- *Delete an entire tree *except* a given node and descendents, replacing the Tree.
- *Obtain a list of a node's descendents.
- *Obtain a list of all leafs (childless nodes) beneath a node.
- *Visit all descendents or all descending leafs and call a specified method on them (for data collection or modification). This is where the real work is done.

All recursion is depth-first.

As an example of what can be done, I made a toy "TicTacToe" class which populates the entire game tree for this simple game. I visit all endgames (leafs), and collect win/loss statistics. This tree had around 350,000 nodes.

If you have data which is naturally organized heirarchically, this may be useful for you.

```
J.D. Smith |*| WORK: (607) 255-5842
Cornell University Dept. of Astronomy |*| (607) 255-6263
304 Space Sciences Bldg. |*| FAX: (607) 255-5875
Ithaca, NY 14853 |*|
;+
; NAME: ObjTree
;
; PURPOSE: A Object Based data Tree
;
; CATEGORY: Object-Based Data Manipulation
;
: METHODS:
```

CHILDREN(): Return this node's child object(s).

SIBLING(): Return this node's sibling object(s)

SETVALUE: Sets Object Data Values -- use ADD unless custom addition. KEYWORDS: Each keyword sets the corresponding member data with the passed value, which can either be an object list or pointer to one.

CHILDREN: The node's children

SIBLINGS: The node's siblings ... points also to parent's

children list.

LEAFS, list: return a list of all leafs below this node.

FAMILY, list: return a list of all of this node's descendents.

VISIT, method: Recursively call method method on all descendents. passing it any extra keywords (which might be required to effect any modification of the node data).

LEAFVISIT, method: Same as visit, but only for leafs.

ADD, entry: Add the object entry to this node.

KEYWORDS:

SIBLING: Add entry as a sibling instead of default child.

NOTE: An added sibling is always younger than any existing siblings. An added child inherits is made the sibling of any existing children.

DELETE: Delete this node, and all of its desendents (recursively). NOTE: If this node has a parent, its child reference is assigned to the next younger sibling, if no older siblings exist. If no other siblings exist at all, the parent's child reference is cleared.

PRUNE: Delete the entire tree *except* this node and its descendents, leaving this node as the root of the Tree.

CLEANUP: (Automotically called)

NOTES: Each node in the tree is represented by a single instance of this class. This tree has these properties: A node's list of children is the same list as its children's list of siblings.. not a copy, the *same* list -- modifying the children list at the same time modifies the children's sibling list. New generations can only be added at extremities of the tree, i.e. at those nodes which don't yet have children. Otherwise, new children will join those children already living. You may use SetValue to circumvent these limits, but beware: inbred trees may result.

```
MODIFICATION HISTORY:
 11/13/98 -- Added Leafs and LeafVisit. JDS
    6/4/98 -- Removed Gen, changed ModPro to a Call_Method in Method. JDS
  5/12/98 -- JD Smith
function ObjTree::Children
 return, self.children
end
function ObjTree::Siblings
 return, self.siblings
end
pro ObjTree::SetValue, CHILDREN=child, SIBLINGS=sib
 if keyword set(child) then begin
   ;; find out if list or pointer is passed
   s=size(child,/TYPE)
   if s eq 10 then begin
                       ;it's a pointer
    if child ne self.children then begin ;points at different heap vars?
      ptr free, self.children; free memory of old list
      self.children=child
     endif
   endif else begin
    if ptr_valid(self.children) then *self.children=child $
     else self.children=ptr_new(child)
   endelse
 endif
 if keyword_set(sib) then begin
   s=size(sib,/TYPE)
   if s eq 10 then begin ;it's a pointer
    if sib ne self.siblings then begin ;point at different heap vars?
      ptr_free, self.siblings
      self.siblings=sib
     endif
   endif else begin
     *self.siblings=sib
                      ;siblings must be valid, since *we're* alive.
   endelse
 endif
end
Leafs - Find all of the Leaves below me
pro ObjTree::Leafs,list
```

```
c=self.children
 if ptr valid(c) then begin
   for i=0,n_elements(*c)-1 do (*c)[i]->Leafs, list
 endif else begin : I am a leaf!
   if n_elements(list) eq 0 then list=self else list=[list,self]
 endelse
end
    Family - Recurs over my decendents and return a list of them.
pro ObjTree::Family,list
 c=self.children
 if ptr_valid(c) then begin
   if n elements(list) eq 0 then list=[*c] else list=[list,*c]
   for i=0,n_elements(*c)-1 do (*c)[i]->Family, list
 endif
end
 Visit: Recurs over my descendents, modifying the node data with a
 method "Method" (presumably of an inheriting class). Any keywords
 passed are given directly to Method to do with as it pleases
 (though in general it will modify or collect data). As a simple
 example, suppose each node had some data member which needed to be
 incremented. A method 'Increment' could do this, and be passed the
 INCREMENT to perform (as a keyword).
  e.g. thisNode->Visit, 'Increment', INCREMENT=10
 Or maybe you need to collect some data, with
  e.g. thisNode->Visit, 'DataCollect', OUTDATA=out
 for putting a summary of data into "out" (_REF_EXTRA is employed).
pro ObjTree::Visit,Method,_REF_EXTRA=e
 if ptr valid(self.children) then $
  for i=0, n_elements(*self.children)-1 do begin
   Call_Method,Method,(*self.children)[i],_EXTRA=e
   (*self.children)[i]->Visit, Method, EXTRA=e; recurs, depth first!
 endfor
end
______
______
: LeafVisit: Same as Visit except on underlying leafs only.
```

```
pro ObjTree::LeafVisit, Method, REF EXTRA=e
 if ptr_valid(self.children) then begin
   for i=0, n_elements(*self.children)-1 do $
    (*self.children)[i]->LeafVisit,Method, _EXTRA=e
 endif else begin
   Call_Method, Method, self, _EXTRA=e
 endelse
end
 Add: Add element(s), as either children or siblings (children
 by default), to the current node (this object instance).
 Siblings added are always younger (later in list) than any existing
 siblings.. and children added are assigned to be younger than
 any children already present, and at their depth. This means
 that new generations can only be created at the bottom of the
 tree (if the root is at the top).
pro ObjTree::Add, list, SIBLINGS=sib
 if keyword_set(sib) then begin ;inserting new sibling(s)
   ;; add them to the end of my list
   *self.siblings=[*self.siblings,list]
   ;; set their siblings array the same as mine, freeing any old siblings
   ;; list if any (should't have any, inbred trees are trouble!)
   for i=0,n elements(list)-1 do $
    list[i]->SetValue,SIBLINGS=self.siblings
 endif else begin
                        ;inserting a new child
   if ptr valid(self.children) then begin
     ;; add the children at the end of my children list
     *self.children=[*self.children,list]
   endif else begin
     self.children=ptr_new(list)
   endelse
   ;; set their *siblings* array be my *children* array, freeing any
   ;; siblings list if any (but there shouldn't really be).
   for i=0,n elements(list)-1 do $
    list[i]->SetValue,SIBLINGS=self.children
 endelse
end
Delete: Delete this node and all descendents, clearing
; the child list of its parent if this node has no siblings.
```

```
pro ObjTree::Delete
 sibs=n_elements(*self.siblings); if only 1, I'm an only child.
 if sibs eq 1 then self.siblings=ptr_new() else $
 *self.siblings=(*self.siblings)[where(*self.siblings ne self)]
 obj_destroy,self
                     ;call cleanup recursively to kill descendents
end
; Prune: Delete everything in Tree except this node and it's
; descendents, leaving this node as the root of the Tree.
______
pro ObjTree::Prune, Tree
 ;; remove myself from my list of siblings (and from my parents list of
 :: children -- it's the same list!!)
 if n_elements(*self.siblings) gt 1 then $
 *self.siblings=(*self.siblings)[where(*self.siblings ne self)] $
 else $
 self.siblings=ptr_new()
                      ;I was an only child
 ;; Destroy the tree around us, as we hide, not on the list for destruction.
 obj_destroy,Tree
 Tree=self
                   :I am now the root of this tree!
end
 ------
Cleanup: Recursively destroy all descendents, depth first.
pro ObjTree::Cleanup
 ;; Call Cleanup on children first, then cleanup the siblings list (which
 ;; will also free the children list of the siblings' parent).
 if ptr valid(self.children) then begin
   obj_destroy,*self.children
 endif
 if ptr_valid(self.siblings) then ptr_free,self.siblings
end
ObjTree__define: define the ObjWidget Class structure
pro ObjTree define
 :; define a tree member class
```

```
struct={ObjTree, $
       siblings:ptr_new(),$; an array of siblings (at least including me!)
       children:ptr_new(); ;an array of children (possibly childless)
end
```

File Attachments
1) objtree__define.pro, downloaded 90 times