ON DISK

Object-Oriented Display Refreshing

CallLists is a refreshing way to keep your interactive programs up to date.

By Bryan Ford

INTERACTIVE PROGRAMS (containing complex user interfaces) pose problems that you don't have to deal with in batch processing programs. When writing a batch program, such as a compiler, you process a complete set of input data in a certain format and write it out in another format. An interactive program, on the other hand, has to process input data that comes from the user in little pieces at a time (events). These events usually are entered in no particular order, and often change or even reverse the results of previous events. An interactive program must generate new, updated output very soon—no user likes to wait for a slow program—after each input event.

This article deals with one of the main problems in writing interactive programs: refreshing the various parts of a program's display at appropriate times. Some programs have many different windows, possibly with several separate display areas in each window. When some part of the project is changed (a new event is received), the program must immediately update the appropriate windows to reflect the changes.

Probably the simplest method of refreshing displays is to create one big function that calls all other screen refresh functions one by one:

```
void RefreshAll()
   {
      RefreshViews();
      RefreshQuickMenu();
      RefreshThis();
      RefreshThat();
   }
```

This method is very simple, and assures that *everything* is up-to-date after it has been called. While this method works well for simple requesters or very small programs, once you add a more windows or other display areas, the program becomes slow and flickery.

On the opposite extreme, you could call the appropriate refresh functions immediately after each change is input. Although this method can eliminate unnecessary refreshing, it requires you to remember all of the program's interdependencies. It also tends to create subtle bugs and make revision difficult at best.

Two other problems are not adequately addressed in either of these methods. First, what if one refresh function depends on another? For example, in a 3-D modeling program, DrawPolygons() might depend on CalculatePolygons(). These functions must be called in the correct order to ensure that the final display is completely up-to-date—if you draw objects before you recalculate their positions, the display will always remain one step behind the rest of the program. A reliable method must be found to ensure the correct calling sequence.

The second problem arises when a refresh function is called many times quickly. For example, a user might select several menu-based commands at once, each of which performs some simple operation on the project. A poorly written program will often refresh the display after each operation, even though it really only needs to be refreshed once at the end. A more common example is repetition of an operation—how many times have you briefly held down some repeating key in a program and had to wait a long time afterwards for it to finish processing your keystrokes?

These problems can be solved using various combinations of functions and function calls. Using flags to keep track of items that need refreshing can also help. No matter how carefully you lay out the function calls and flags, however, you will still run into problems as the program gets bigger.

A BETTER WAY

A cleaner and more efficient method of handling refreshing is "object-oriented refreshing," which uses data structures and lists rather than direct function calls. As you will see, it can conveniently solve all of the problems I've mentioned using a small amount of extra code. It also helps prevent obscure bugs and makes program revision simpler.

The basic data structure used by the system is as follows:

```
struct CallNode
   {
      struct Node Node;
      LONG (*CallFunction)(LONG GlobalData, LONG LocalData);
      LONG LocalData;
   };
#define CNT_INLIST NT_USER
```

A CallNode represents a function that must be called sometime in the future. Think of it as a "computer agenda item." As you can see, it is based on Exec's standard Node structure and is meant to be linked into a standard Exec MinList (the stripped-down version of the standard List).

When an event occurs causing a particular part of the display to need refreshing, a CallNode is added to a List (a CallList). Instead of calling the refresh function immediately, the call is "remembered" for later. After adding the CallNode, the program can continue processing other events immediately.

A typical program will have one global CallList in the main module and many small CallNodes defined in other modules, each one representing a particular refresh function that might be called at some time. For example: ►

```
static void RefreshQuickMenu(long globaldata,long localdata)
  {
    (do all refreshing of the quick menu)
  }

struct CallNode CallQuickMenu = {
  {0,0,0,<priority>},RefreshQuickMenu,<local data>};
```

Then, wherever the quick menu is changed (or for some other reason needs refreshing), instead of immediately calling RefreshQuickMenu(), just make this call:

```
AddCallNode(&RefreshCallList,&CallQuickMenu)
```

(If a particular CallList is used often, you might want to define a macro that adds CallNodes to a specific CallList.) This adds the CallNode to the specified CallList. The AddCallNode() function is defined as follows:

```
void AddCallNode(struct CallList *l,struct CallNode *n)
  {
    if(n->Node.ln_Type != CNT_INLIST)
      {
        n->Node.ln_Type =CNT_INLIST;
        Enqueue((struct List*)l,&n->Node);
      }
  }
```

When it's time to refresh the display (probably just after all of Intuition's IntuiMessages are handled, but before you call Wait()), you simply make a single call to the function CallRem():

```
long CallRem(struct CallList *l,long GlobalData)
  {
  struct CallNode *n;
  long ret;
    while(n = (struct CallNode*)RemHead((struct List*)l))
    {
      n->Node.ln_Type=0;
      if(ret=(*n->CallFunction)(GlobalData,n->LocalData));
          return(ret);
    }
  return(0);
}
```

This function traverses the entire CallList, removing each CallNode as it progresses, and calls each refresh function in turn. (Note that CallRem(), in the source code in the accompanying disk's Ford drawer, is actually a macro that uses a more powerful version of this function, explained later.) Each CallFunction must return zero if it wants the CallList processing to continue (the usual case) or nonzero if it wants to stop processing the list (if an error occurred, for example).

By now you should have a general idea of how the system operates. The rule of thumb is: Save the refresh function calls until later, and then only call them when the user is no longer doing anything.

MAPPING THE LISTS

The use of the CallNode's ln_Type field prevents any CallNode from being added to a CallList more than once. This helps to prevent corrupting the list, makes the caller's life easier and conveniently solves the problem of queueing up many identical or similar refresh events. You can make as many AddCallNode() calls for a particular CallNode as you want, but the function is only called once at the end, eliminating the need for "dirty" flags and such.

Because Enqueue() is used to add nodes to a CallList, the list always remains sorted according to priority. Because CallRem() starts at the head of the list and works toward the tail, it always calls CallNodes with the highest priorities first to ensure that all refreshing is done in the correct sequence. For example, in our 3-D modeler example, CalculatePolygons() would simply be assigned a higher priority than DrawPolygons() and, therefore, would always be called before DrawPolygons().

As you set up your CallLists and CallNodes, keep track of which functions depend on which other functions, and set the CallNode priorities appropriately. I recommend keeping handy a text file that lists all the CallNodes for a given CallList, their priorities, and where in your source code they are located. This will help you assign priorities to new CallNodes and will give you a good overview of which refresh functions are being called, in which order.

In general, CallLists and CallNodes can simply be defined as global or static variables. I generally define a CallList near the corresponding call to CallRem(), and put CallNodes just below the functions they point to. In some cases, however, dynamically allocating CallNodes may be more appropriate. Just make sure you always allocate the memory with MEMF_CLEAR, or at least initialize the Flags field to zero before using the CallNode. You can also have more than one CallRem() call for a given CallList, if you find this beneficial. Just remember that each call to CallRem() completely empties the list, so the next call won't do anything unless you first add additional CallNodes.

"As you set up your CallLists and CallNodes, keep track of which functions depend on which other functions, and set the CallNode priorities appropriately."

Any CallFunction (a function pointed to by a CallNode and called by CallRem()) may make calls to AddCallNode(), even on the CallList that's currently being traversed. This provides a convenient solution to another dependency problem. In our 3-D modeler example, CalculatePolygons() calls AddCallNode() with the CallNode for DrawPolygons(), forcing a redraw to occur sometime after any recalculation, without either of the functions actually calling the other directly. In general, you should add other CallNodes with lower priority only this way, although it is safe to add any CallNode if you are careful about interdependencies.

MORE SYSTEM FEATURES

For flexibility, the system provides a general-purpose mechanism for passing parameters to the CallFunctions. The GlobalData variable is a LONG (you may use it as a pointer if you wish) that is given to CallRem() and passed through to all of the functions on the CallList, as they are being called. This provides a convenient way to "broadcast" a piece of data to all the called functions. Similarly, each CallNode contains a LocalData variable that is passed to the CallFunction when it is activated. This is particularly useful if you dy-

Continued on p. 63

From p. 28

namically allocate CallNodes—the LocalData might be a pointer back to the CallNode or to some other data structure associated with this particular CallNode.

There are several other functions provided in the library that add additional functionality to the system and make it useful for applications other than refreshing.

The function RemCallNode() is exactly the opposite of AddCallNode()—it removes a CallNode from a CallList without actually calling the function. Simply call it as you would Exec's Remove() function. As you can AddCallNode(), you can safely call this function with a CallNode that was not previously on any list. (See the source code on disk for the actual definition of this function.)

Finally, if you want to call all of the CallNodes in a CallList, but leave the list intact so you can later use the same list again, just use the Call() function. In this case, an explicit call to RemCallNode() is required to remove a CallNode. This will generally not be useful for refreshing, but may be useful in other applications of this system.

If you use this variation, you should not call AddCallNode() from within the CallFunctions, because newly added nodes may be accidentally skipped (even if they have a lower priority than the current node). You should also not call RemCallNode() on *other* nodes in the same list from within a CallFunction. However, you may RemCallNode() the *current* node from within its own function, to indicate that that function doesn't need to be called anymore.

There are also two variations of Call() and CallRem(), named CallExt() and CallRemExt(), respectively. These variations accept the same arguments as the regular versions, plus an extra parameter: a pointer to a function to call after each CallFunction on the list is called. You can use this facility, for example, to check for such signals as CTRL-C while the list is being processed. See the source code on disk for more details about these functions.

By now, you have probably noticed that CallLists can be used for applications other than display refreshing. For example, during your program you can maintain a CallList that is called only once, when your program is about to exit. When a module in your program allocates some memory or opens a window, it simply adds a CallNode to this global CallList. When the program is about to exit, this CallNode will automatically be called, so the module can free anything it previously allocated.

Using CallLists can help make screen refreshing elegant and efficient. Unnecessary processing is eliminated and interdependencies can be handled easily. Because the CallList library functions use no global variables, they can be reused as much as necessary in one program, and will cause no problems (if used correctly) in re-entrant code such as run-time libraries. Remember, CallLists are useful for many purposes, display refreshing is just one of them. ■

Bryan Ford is a student at the University of Utah and works on freelance programming projects for local companies. Contact him c/o The AmigaWorld Tech Journal, *80 Elm St., Peterborough, NH 03458, or on Internet (bryan.ford@m.cc.utah.edu).*