MUJ
MACROMEDIA USER JOURNAL™
http://www.muj.com
March 1999
Number 80

Director Essentials: New Text Features and More "How-to" Lingo

Kevin McFarland

The February 1999 installment of this "Director Essentials" series covered the new text features in Director 7, generally acclaimed to be one of the best Director revisions in quite some time. This article will continue with more "how-to" Lingo, covering user text entry. As in previous issues, the attempt is to cover features commonly requested by clients—ones that every beginning and intermediate Director user should be familiar with. Hopefully, even advanced users will find some new tidbits of information!

THE text engine in Director 7 has been completely reworked. At first glance, text cast members may appear to be no different. You can still create them in the same manner, and when you place them onscreen, they may appear to be no different. However, a closer inspection reveals many differences and some great new improvements (along with a few things to watch out for).

Cast members created with the text window in Director 7 are, fundamentally, a new *type* (literally) of cast member. To see this for yourself, open Director 6, create a new text cast member in the text window (Ctrl-6 on the PC, Cmd-6 on the Mac), name the cast member "sample", and then use the message window to ask Director what type of cast member it is:

```
-- Director 6
put the type of member "sample"
-- #richtext
```

Now try the same thing in Director 7, and notice the different result you get:

```
-- Director 7
put the type of member "sample"
-- #text
```

(or, using Lingo's new "dot" syntax):

```
put member "sample".type
-- #text
```

What this means is that #richtext cast members no longer exist—they've been replaced by this new #text type of cast member. (Note: For the remainder of this article, I'll refer to this cast member type in the same way Lingo does—by putting a pound sign in front of the word "text,"—to differentiate them from #field members, which also contain text, but are a very different type of cast member.)

These new #text cast members have three components: text, rich text (RTF), and hypertext markup language (HTML). Again, if you were to create a sample cast member in Director 7 and then type the following messages into the message window, you might see something like this:

```
put member("text sample").text
-- "This is a test"

put member("text sample").rtf
-- "{\rtf1\ansi\deff0
{\fonttbl{\f0\fnilSystem;}{\f1\fswissArial;}}
{\colortbl\red0\green0\blue0;\red0\green0\blue224;\red224\
green0\blue0;\red224\green0\blue224;}{\stylesheet{\s0\fs24
NormalText;}}\pard \f0\fs24{\f1\fs36 This is a
test\par}}"

put member("text sample").html
-- "<html>
<head>
<title>Untitled</title>
</head>
<body bgcolor="#FFFFFF">
<font face="Arial, Helvetica" size=5>This is a test
</font></body>
</html>
"
```

In other words, as you type, Director creates the three different types of information found in a #text cast member: text, rich text, and HTML (an HTML generator is built in!).

Each one of these three components of the cast member is accessible and can be modified via Lingo; furthermore, changes to any one component (either manually or via Lingo) will update the other two on the fly.

New things you can do with text in Director 7

The advantages this new #text cast member type offers over the old #richtext cast members found in Director 5 and 6 are significant. These new #text cast members:

- are dynamic—they can now be manipulated by Lingo at runtime;
- can be edited by the user at runtime (anti-aliased editable text!);
- can be animated using Director 7's new animation effects (e.g., rotated);
- can be any size (they're no longer limited to 32K);
- use less memory;
- have built-in hypertext capabilities; and
- can be imported from HTML-formatted docs with HTML formatting retained (basic text only; no pictures or complex table tags).

Changes you'll want to be aware of

The biggest change is that non-system fonts used by #text members must now be embedded in a Director file. This is done by going to the Insert menu and choosing Insert: MediaElement: Font. (See Figure 1.)

Obviously, the font to be embedded must be installed on the machine at the time of embedding. After embedding, these fonts live in the cast as actual cast members, and #text members depend on this embedded font information to display. This is different from earlier versions of Director, where the old #richtext members basically became bitmaps at runtime and so didn't depend on any font information.

Another change to be aware of is that #text cast members can require significantly more processing power to display. They now include a "direct to stage" option in their cast member info dialog boxes. (See Figure 2.)

Like the direct to stage option for video cast members, selecting this option will render the text directly to the screen rather than to Director's offscreen imaging buffer, thus speeding up display, although you may still notice a performance hit (see "Problems you'll want to watch out for" later in this article). And also, like video cast members, selecting this option means you can't have any sprites overlaid on top of the direct to stage member, and that member will be rendered onscreen using copy ink, no matter what ink you have it set for in the score.

Figure 1. Embedding fonts in a Director file.

There are some Lingo issues to be aware of as well. The fact that #text members now contain more information (RTF, HTML) means that attempting to manipulate certain parts of that information via Lingo can sometimes be more complex than manipulating a simple field. Also, features new to Director 7 will respond only to Director 7's new dot syntax. #Text members are a new feature, and so some Lingo used with #text members will probably fall into this category.

Here's an example. With a #text member, Director 7 chokes on this syntax:

```
set the fontStyle of line 1 of member("test") = "bold"
```

but easily digests the following:

```
member("test").line[1].fontstyle = [#bold]
```

#Field cast members still exist for purposes of backward compatibility with older projects, and also because they're still the best choice for large blocks of text that can be displayed in system fonts and do not need to be anti-aliased, as they require less processing power to display and manipulate. (Also, like #textmembers fields, they can now display themselves using embedded font information and are no longer limited to 32K in size.)

Speaking of backwards compatibility, when older Director files are updated to Director 7, #richtext members will be converted to #textmembers. It's important to remember that the font used by those #richtextmembers must be installed on the machine at the time of updating.

Problems you'll want to watch out for

It's highly unlikely that there will be no bugs or "issues"

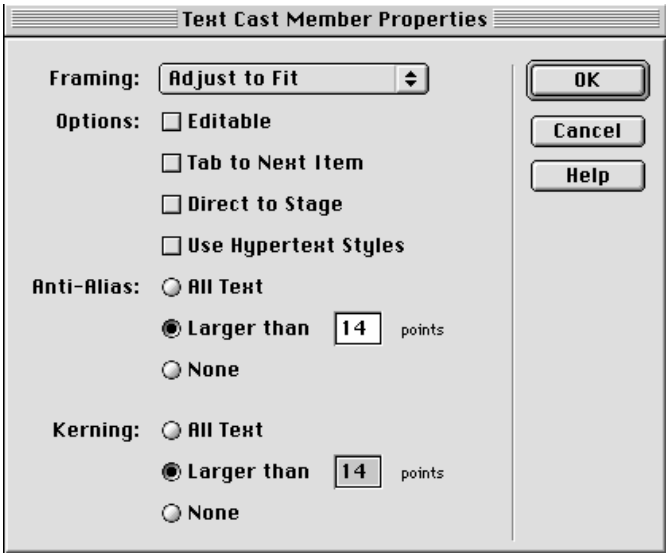

Figure 2. The direct to stage option for #text cast members.

with this newest version of Director in general and with this new type of #text member in particular. Director is a complex software product that by its very nature has to interface with many different types of media, thus increasing the likelihood of problems in some interactions. If history is any judge, we'll probably see a 7.0.1 version at some point in the next few months as new bugs and issues are discovered and (hopefully) fixed. As always, caution and testing are advisable when deciding whether to release a product immediately using this new version of Director.

Major problems with #textmembers (as of early 1999), both from my own experience and as reported from sources such as mail lists and Web sites, include:

- The increased processing power required to render #text members to screen may result in a noticeable performance hit if there's animation occurring simultaneously as #text cast members are displayed. This can be somewhat improved with the new "direct to stage" option; however, as mentioned previously, you're limited to copy ink effects when using this option. (Performance, of course, will also be improved by preloading the #text cast members into memory.) In informal testing, even when I opened a new "bare-bones" .dir file and created a simple film loop and a few #text sprites, I personally experienced varying results on different machines with using the direct to stage option, especially on lower-end processors.
- Some TrueType fonts don't render their bold or italic styles correctly onscreen. To the casual eye (like mine), they might appear to be okay; however, to the trained eye (like the designer I'm working for, who was one of the first to alert me to this issue!), some characters aren't correctly represented. The problem appears to be limited to TrueType fonts.
- There are reports of crashes when importing and then attempting to edit RTF or HTML text files larger than approximately 8K.
- There are reports that large numbers of #text members (over 45) in a frame, set to background transparent ink, cause the #text members to become invisible on stage.
- Incorrect results have been reported when querying a #text member that contains more than approximately 29 lines (e.g., "the lineCount of member...").

Minor issues (noted in Director 7's ReadMe file) include:

- RTF text files do not split into separate members at page breaks when imported.
- On the Macintosh, a small percentage of PostScript fonts either fail to compress or render incorrectly.
- Under some circumstances, text may be rendered in an incorrect font. Do a Save As and restart Director.
- Text that disappears when a movie is updated may have a blend value of 0 (Director 6 ignored blend values for text sprites).
- Chinese and Korean fonts are not anti-aliased and cannot be embedded.

"How-to" Lingo for working with user-entered text

Here are some "how-to" solutions for common tasks in Director involving user text entry, a.k.a. "editable" text. (Like many users, I'm used to referring to text entry areas as text fields. In Director 7, these might or might not be actual #field type members. In the following examples, don't take the use of the word "field" to mean an actual #field member; in Director 7, it could just as easily be a #text member.)

How can I allow the user to input text?

Typical cases where this might be desired include allowing the user to input a name, ID, password, data, personal information, notes, and so on. Once entered by the user, the text can be manipulated via Lingo, saved to disk, and otherwise operated on just as any other text string might be.

To allow user entry into a text member, you need to set the "editable" property of either the sprite or the member. In Director 7, you can set this property for either #text or #field members, or sprites containing those types of members. Users of earlier versions are limited to setting this property for #field members (or sprites containing fields) only.

You can set this property either with or without using Lingo. For the former, select the sprite and then select the "editable" checkbox in the score window (note that this checkbox has a slightly different appearance in Director 7); or select the member in the cast window, click on the cast member info button, and select "editable" in the dialog box. To set this property via Lingo, you can either set the editable of the sprite (the following example uses sprite 3):

```
sprite(3).editable = true
```

or the editable of the member itself:

```
member("sample").editable = true
```

Users of Director 6 or earlier will need to use "old" style (non-dot) Lingo syntax:

```
set the editable of sprite 3 = true
```

or:

```
set the editable of member "sample" = true
```

Should you set this property for the sprite or for the member? Obviously, setting the property for the member would mean it would always be editable, as opposed to only being editable for the duration of the sprite's span in the score. See the next "how-to" for an example of this. (Side note: In Director 6, I've experienced subtle differences in whether the editable text was selected or not when going to a frame containing that member, depending on whether the "editable" property was set for the sprite or for the member.)

How can I display the user's name or other data later on in my program, after they've entered it?

For example, you might want to display some sort of welcome message that includes the user's name that was

previously entered. A simple, albeit primitive, way to do this is to set the "editable" property of the sprite, not the member itself (as described previously). On the screen where the user enters his or her name, select the sprite containing the editable member and then select the "editable" property in the score window. At a later screen where the user's name is to be displayed, that same member can be placed onscreen (e.g., after the word "welcome"), except this time with the sprite's editable property turned off.

This, of course, might not give the desired artistic effect, as you may want to use a smaller font size for the sign-in text field but a larger font size for the "welcome" screen, rather than the visually stunning 12 pt. Geneva, Arial, or similar font you used for the sign-in screen. A better way is to use a second cast member, which is of a larger font size, to actually display the contents of the sign-in field. After the user has entered his or her name, use Lingo to transfer the contents of the sign-in field to the cast member used in your "welcome" screen:

```
member("welcome").text = member("entry").text
```

Or, using non-dot syntax:

```
set the text of member "welcome" = the text of ¬
member"entry"
```

This can be done after the user has pressed the return key or some other button signifying that he or she is finished with the sign-in screen.

How can I detect when the return key (or any other key) is pressed when the user is entering text?

Users may expect that pressing the return key will process their text input. It's user shorthand for signaling "I'm finished typing."

The process of testing for which key the user has just pressed is generally the same no matter what key you're testing for. When the user presses a key, a keyDown event is generated. Write a script to trap for the keyDown event, and then, using Lingo functions such as "the key" or "the keyCode", you can evaluate the key that was just pressed and execute the appropriate action.

Here's how to add the Lingo to check for the return key being pressed. Select the editable text or field member in the score, and attach the following sprite script:

```
on keyDown
  if the key = RETURN then
    -- do whatever should happen
    -- when the return key is pressed
    go next
  else
    -- pass the key pressed on to the editable member
    pass
  end if
end
```

A few notes on the preceding script example:

- Keystrokes are "consumed" by a keyDown event handler and aren't passed on to the editable text member. What this means is that unless you explicitly add a "pass" command at the end of a keyDown script, nothing will show up in the editable text member when the user types.
- You can replace the "go next" command in the preceding script with whatever you want to have happen when the user finishes entering text.
- By placing the above script in a sprite script, you limit the scope of the script to the sprite it's attached to; that is, only that sprite will react to the script. This is usually desirable. If, however, you want a wider scope, you can place keyDown scripts in frame scripts (to have all editable text members in that frame respond) or movie scripts (to have all editable text members anywhere in the movie respond). You can also turn key scripts on and off dynamically by setting "the keyDownScript".
- Notice that there are no quotes around return in the preceding script example. Normally, when checking for keys with "the key" function, you'd do a comparison against a text string—which, since it's a string, would need to be enclosed in quotation marks. If you wanted to test whether the key pressed was "a", for example, the script would read:

```
if the key = "a" then
```

However, some keys, of course, can't be represented by a text string, and so Lingo has a variety of constants (variables set to a constant value) you can use when checking to see whether certain keys such as backspace, tab, and the like have been pressed. What this means in practical terms is that since they're constants, they don't need to be enclosed with quotation marks, as text strings must be. This is why the reference to the return key in the preceding example isn't contained in quotes.

How can I limit user input to numbers only?

Say you have a calculator where the user should only be able to type numbers, or text fields where the user is supposed to enter a phone number, a birth date, or something of that nature. One cumbersome strategy would be to use a long if/then or case statement to check for the key being pressed, and not pass it on to the editable text member if the key wasn't a number. A more feasible approach would be to use "the keyCode" function to check for the numerical key code of the most recently pressed key:

```
on keyDown
  if (the keyCode < 48) or (the keyCode > 57) then
    beep  -- optional
  else
    pass
  end if
end
```

The simplest approach of all, however, is to use the Lingo operator "contains":

```
on keyDown
  if ("0123456789.") contains (the key) then
    pass
  else
    beep  -- optional
  end if
end
```

Note that this also allows the user to enter a decimal character (“.”).

Director 7 includes a “Filter Input Characters” behavior in the “text” section of the library palette that allows you to define which keys the user can enter. (Another feature is that it allows you to force text input to all lower or upper case).

If you want to enable user entry on the numeric keypad, you must check for keyCodes rather than “the key,” and then convert the key code to a string (e.g., “7”), which you’d pass on to the editable member.

How can I limit the number of characters a user can enter?

You might have a small editable text member where the user is only supposed to enter something short, such as a user ID, password, or number of a certain length. One approach that can be used with #field cast members is to select “Limit to Field Size” in the “Framing” dropdown list in the cast member info dialog box. This option isn’t supported for #text cast members, however. Another option in this same dropdown list is “Fixed”; however, this will still add characters to the editable text member, even though they might not be visible onscreen. Using Lingo to check for a maximum number of characters entered might appear at first glance to be a straightforward proposition:

```
on keyDown
  -- limit number of characters in field to 10
  if length(the text of member "entry") = 10 then
    beep
  else
    pass
  end if
end
```

However, what this script doesn’t take into account is the user pressing the delete or backspace keys, or the user selecting a chunk of text in the field with the expectation that the next keystroke will delete that chunk. Once there are the maximum number of characters in the editable text member, no more keystrokes will be accepted, even if the keystroke happens to be the delete or backspace key. You would have to incorporate that functionality into the preceding script by testing for the key = BACKSPACE, and the “selEnd” and “selStart” properties (or, in Director 7, the new “selection” and “selectedText” properties). One way around this extra coding would be to just wait until the user submits the text entry (e.g., presses the return key) to check for the number of characters in the editable member.

How can I allow the user to tab between text entry fields?

Users may expect that pressing the tab key will move the keyboard focus to the next text entry field onscreen. (Focus means which text entry field is selected and ready for user input.)This is easily done in Director by enabling auto-tab (“tab to next item”) in the cast member info dialog box for the editable field or text member. When the keyboard focus is on this member and the user presses the tab key, the focus will be shifted to the next highest sprite in the score that’s set to editable (see the section titled “How can I allow the user to input text?”). Auto-tab will loop from the highest-ordered back to the lowest-ordered editable sprite.

Shift-tab (going backwards through editable sprites), however, doesn’t work with auto-tab. In order to implement this, you need to set keyboard focus yourself via Lingo.

How can I get and set keyboard focus when there are multiple text entry fields onscreen?

Most of the time, you as a Lingo programmer won’t need to worry about getting or setting keyboard focus. It’s set automatically by going to a frame containing an editable member, or by the user auto-tabbing, or by the user clicking on an editable member. However, if you want to enable user keyboard control between multiple onscreen editable fields that involves anything more than simple auto-tabbing between fields, you’ll probably need to work with getting and setting keyboard focus. Examples of this would be if you want to enable shift-tabbing (as discussed) or if you need to track which field has the focus for other purposes, like if an individual popup Help or “tip” box was to be displayed for each field when focus was shifted to that field.

How to get and set keyboard focus in Director is a question often asked by developers, simply because prior to Director 7 there were no direct Lingo commands to accomplish this (even in Director 7, you still can’t easily get focus—that is, determine which field has the focus), and it’s neither intuitive nor easy.

There are two common ways developers have dealt with this problem. One is high-tech but requires more coding, while the other is low-tech and requires more score work, but less coding. Both approaches require that you turn the “auto-tab” and (for pre-Director 7 users) the “editable” properties of editable text members off, and instead handle everything yourself. (In Director 7, you can use the new keyBoardFocusSprite command to set focus, which means you could leave the “editable” properties of editable text members on.)

The “high-tech” approach is to set up some sort of key control script where you’d keep track of which field the user is typing in. You would need to watch for when the user hit a tab key (or return key, or whatever other key you determine should take the user from one field to another), and when that key is pressed, you’d turn off the editable of the sprite the user was just in and turn on the editable of the next sprite, while using a variable(s) to keep track of which field the user is currently in. When testing to see if the key = Tab, you’d also test for “the shiftDown”, which would tell you if the user wanted to shift-tab back to the previous field. If you wanted to allow the user to select a field with the mouse for typing into, you’d also need to watch for and keep track of mouseDown events on those fields. The advantage of this is that it can all be done with one frame in the score.

The “low-tech” approach would be to lay out a series of frames in the score, with each frame corresponding to a different text field. If you had 10 user entry fields on one

Continued on page 12

Two Browsers in One with the ActiveX Xtra, Part Two

Randy Weinstein

In the February 1999 issue, Randy showed how to build a dual browser in Director. Now he shows how to do the same thing in Authorware.

YOU can use ActiveX controls in both Director and Authorware. Every feature that we put in the dual Web browser written in Director in the previous issue can be implemented in its sibling authoring system.

I'd be lying if I said the implementation was done with equivalent ease: The Authorware version required substantially more elbow grease, but it worked. Although Authorware and Director dictate very different approaches to multimedia development, using Xtras and ActiveX controls often—though not always—allows us to achieve the same results.

That's because Xtras and ActiveX controls *should* behave the same way no matter which authoring environment they're used in. They're a lot like vending machines: They have physical features and procedural methods for making requests of them (i.e., pushing buttons and sticking in money), and they have various ways of responding to your actions (i.e., delivering soda pop, making change, asking for more money). And if you take a candy machine out of the local Laundromat and put it in the neighborhood grocery store, your interactions with it will remain essentially the same. Of course, you need to understand how it operates if you want to get something from it, regardless of where it's located. The same thing applies to an ActiveX control: It's important for you to know how to manipulate its various components (i.e., methods and properties) and to interpret the messages it sends back to you in order to make it behave the way you want.

Building the Authorware Web browser

To use the Web browser control in Authorware, you follow the same steps as in the Director version. In addition to knowing how to work with the control itself, you have to know how Authorware works with the control. You can get help on the Authorware issues by using the manual and the Help pages. To understand how the Web browser control works, I recommend you study http://www.microsoft.com/workshop/c-frame.htm#/workshop/browser/default.asp on Microsoft's Developer Network site. This page documents the methods, properties, and events generated by the Web browser control.

Before we start rebuilding our double-duty browser in Authorware, let's assume the following:

- The machine you'll be using is a Windows 95, Windows NT, or Windows 98 computer.
- The machine supports ActiveX.
- The machine has the latest version of the Microsoft Web browser ActiveX control properly installed and registered.
- Both ActiveX.X32 ActXPriv.X32 are installed in the Xtras subfolder of your Authorware program.
- The machine is currently connected to the Internet.

The first thing you need to do is incorporate the ActiveX Xtra into the Authorware file. Position the Paste Hand on the flow line. Then click on "Insert" in the Authorware menu, and then select the "Control" submenu. The ActiveX control will be listed there. (See Figure 1.)

Once you select the ActiveX option, the ActiveX Xtra dialog box appears. Select the Microsoft Web Browser Control from those listed. Authorware will then insert an icon on the flow line called "ActiveX..." which you can rename. To add another browser, simply repeat the procedure just described.

As with Director movies, we can distribute Xtras with our Authorware titles, and as long as they reside in an Xtras subfolder of the title's folder, things should work fine. There's an additional configuration issue involved when using the ActiveX Xtra: We need to make sure that the machine on which the title runs supports ActiveX and that the Web browser control is installed. We'll use the

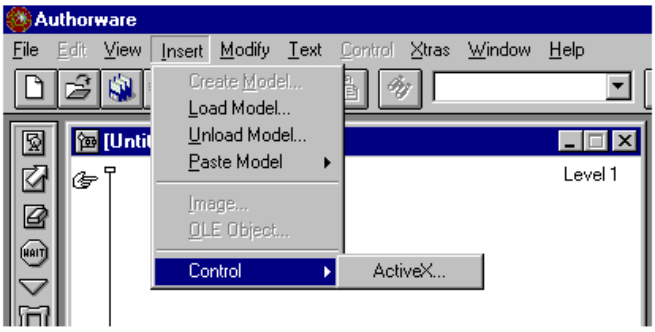

Figure 1. Incorporating the ActiveX Xtra into the Authorware file.

same ActiveX Xtra methods used previously, but we'll use a combination of Decision and Calc icons to validate the operating environment. For clarification, understand that we have to use the ActiveX Xtra to interface with the ActiveX control. Sometimes we'll use methods or access properties that belong to the Xtra itself; at other times, we'll be dealing with methods, properties, and events belonging to a specific ActiveX control.

The *ActiveXInstalled()* method of the ActiveX Xtra determines whether or not ActiveX is installed on the computer on which the program is running. If ActiveX is installed, it returns the value 1; if not, it returns a 0. If you set the Decision icon branch type to calculated path and set the calculation to ActiveXInstalled() = 0, the Decision icon will branch to the first icon on its path. Since it's true that ActiveX isn't installed if the ActiveXInstalled() function returns 0, and since a true condition is numerically represented as a value of 1, the branch path for the Decision icon is thus set to 1. (See **Figure 2**.)

I put a Map icon on the branch path that contains a Display icon informing the user that ActiveX control is not supported on their machine, a Wait icon, and a Calc icon that contains Authorware's Quit function to shut down the program. (See Figure 2.)

This same model is used to determine whether or not the Web browser control is installed using the ActiveX Xtra method *ActiveXControlQuery()*, which searches for a particular control by looking up its class identification in the system Registry. Every ActiveX control has a unique class identifier made up of a really long string of numbers, characters, and dashes. You can find the class ID of a selected ActiveX control by clicking on the URL button on the ActiveX Control Properties dialog box. This brings up another dialog box, and the class ID is listed at the top. Because this ID is kind of unwieldy, I first create a Calc icon and add a statement that assigns the ID of the browser control to the custom variable *BrowserID*. Then I create another Decision icon, set it to branch to a calculated path, and set the calculation to ActiveXControlQuery(BrowserID) = 0. The Map icon is identical to the one attached to the Decision icon previously described, except the Display icon tells the user that the Web browser control is not installed. (See Figure 2.)

Now we're ready to initialize the Web browsers with their initial Web pages. This can all be done in one Calc icon. As we did in the Director movie, we use the Web Browser control's *Navigate* method. However, in Authorware, we have to access these methods in a more roundabout way. Functional objects such as ActiveX controls are represented in Authorware as sprite Xtras. To access a sprite Xtra's methods, you must use the CallSprite function, which has the following form:

```
CallSprite(IconID@"SpriteIconTitle", #method [, ¬
argument ...])
```

We use the CallSprite function twice in a Calc icon following the two Web browser control icons on the flow line. Here's the code:

```
ApolloPage := "http://msdn.microsoft.com/developer/
sdk/inetsdk/help/itt/IEProg/WB/Objects/
WebBrowser.htm"

DionysosPage := "http://www.prs.org/books
/book333.htm"
CallSprite(IconID@"Apollo",#Navigate,
ApolloPage)
CallSprite(IconID@"Dionysos",#Navigate,
DionysosPage)
```

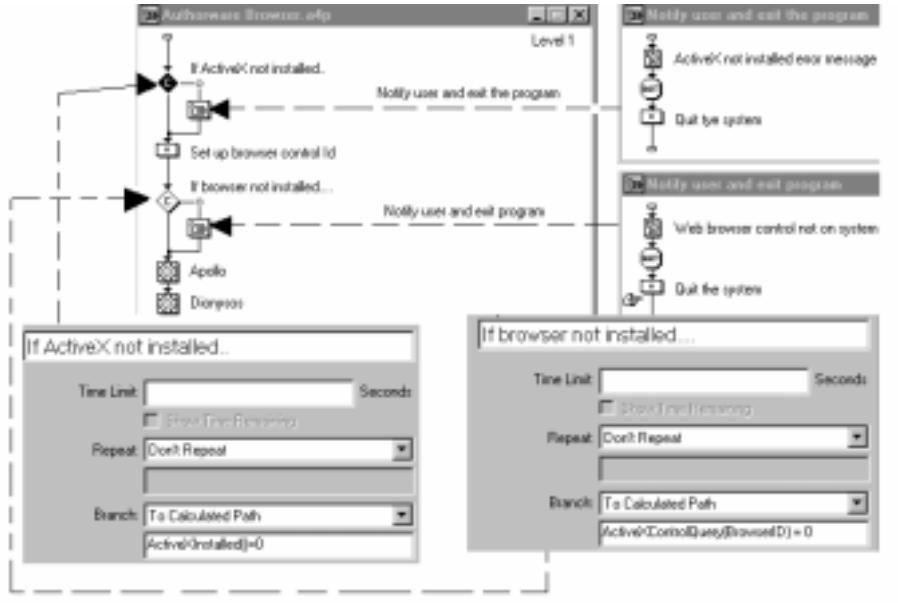

Figure 2. Adding the browser control to the ActiveX Xtra.

Handling events in Authorware

There's one more step needed to duplicate the features put into the Director-based browser: Maintain a list of Web pages accessed during a session. This list is created by intercepting *StatusTextChange* messages sent by the Web browser controls to Authorware and selecting those messages indicating that a new page has been accessed. Authorware can handle messages using what are called *event responses*. In Authorware, a response is part of an interaction between the Authorware title and some external event, which can range from the user clicking a button or pressing a key to a Web browser control telling Authorware that it's downloading a new Web page.

The first step in building our Web page history list is to associate an interaction icon with an event response. In the Authorware version of the Web browser, I place an interaction at the very top of the piece, attach a Calc icon to it, and choose Event as its response type when the Response type selection dialog box comes up. I also set it to be a perpetual response. Once the Calc icon is attached, clicking on the response symbol ("E") brings up a dialog box for setting up the response. (See **Figure 3**.) The Sender list box should list the names of both of the browser controls. Double-clicking on either one will immediately populate the Event Name list box with all of the events the control generates. Double-clicking on the StatusTextChange event will cause this particular event response to respond to the selected event generated by the selected Web browser control. Repeat this process for the other Web browser control.

After establishing a way to capture these messages, we need to examine the text of the newly generated message. Unfortunately, this turns out to be a more roundabout process than one would wish for. For reasons unknown, the Web browser properties don't seem to be consistently accessible using the GetSpriteProperty function. Consequently, I had to resort to looking at a system variable called EventLastMatched to get the data I needed to build the browser history list.

This variable contains a property list for the Xtra event matched at the last event response and provides information about the event sender's ID number, the title of the Sprite Xtra icon sending the event, the icon ID of the sender, the actual event name, and the number of parameters passed back with the event. It finishes with values of each parameter passed back. Here's what this variable, sent to our Web browser, looks like:

```
[#__Sender:79960428, #__SenderXtraName:"Xtra ActiveX",
#__SenderIconId:65633, #__EventName:#StatusTextChange
, #__NumArgs:1, #Text:"Done"]
```

The only value we're interested in is the very last one: the #Text property. This is where the StatusTextChange event will store messages needed to update our history list. In the previous example, the last event matched by the event response to the StatusTextChange message was the word "Done". We know that a new page is being downloaded when the status text begins with the words "Opening page". Thus, we can examine the value of the #Text property. If it contains the words "Opening page", we can extract the URL of the Web page it's currently opening and append it to our history list. Here's the code:

```
temp := ValueAtIndex(EventLastMatched,6)
if Find("Opening page",temp) > 0 then
  temp := SubStr(temp,14,CharCount(temp))
   HistoryList := HistoryList^Return^temp
end if
```

For StatusTextChange events, the sixth element (i.e., the #Text property) in the EventLastMatched variable contains the data we're interested in. This is stored to a custom variable called temp.

The Find function returns the position at which it found the pattern searched for in the indicated text string. If it doesn't find the pattern, it returns 0. If a value greater than 0 is returned, we pare down the temp variable so that it only contains the URL just opened. Temp is then appended to the custom variable HistoryList, which is displayed via text contained in the display of the interaction icon to which the event is attached.

Versions of this Calc icon are attached to both event responses. The only difference between them is the name of the icon supplied to the GetSpriteProperty function.

Pros and cons of Authorware vs. Director

We've now created two versions of the same twin Web browser. In retrospect, I think this particular control is better supported by Director. It's not as cumbersome to analyze events and their associated parameters. Moreover, Web browser control properties are more readily accessible in Director. One of the caveats of using ActiveX controls is that there's no steadfast guarantee that they will work seamlessly with Authorware or Director or that all features of the control will be accessible from both authoring environments. Though imperfect, I'm confident that using the Web browser control is a heck of a lot easier than trying to build a Web browser from scratch—and a lot more impressive than telling a client that you don't know how to add HTML / Web support to a multimedia title! ▲

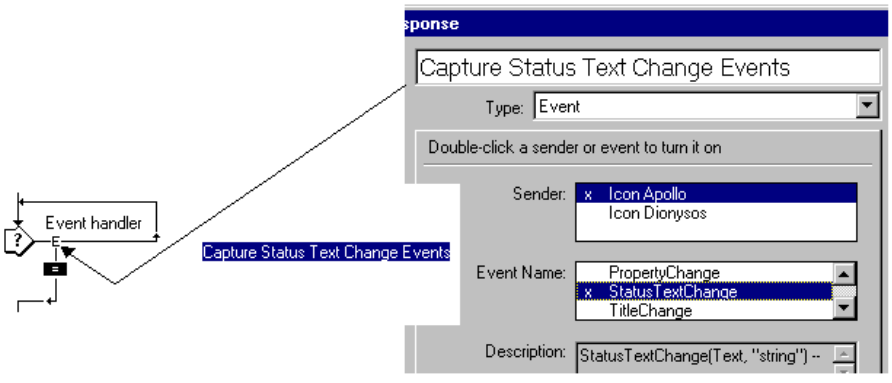

Figure 3. Making the event handler work.

Randy Weinstein is a professional musician and multimedia developer who specializes in computer-based training development and CGI programming. He currently lives in New York. 212-675-5920, sigmonky@mail.well.com.

Quad Squad: Groovy Effects Using Director 7's Quad Sprite Property

Larry Doyle

In the February 1999 issue, Larry lured us to the lusty new features of Director 7. In this article, he reveals how the quad sprite property can create mind-blowing animation effects by distorting sprites on the stage. Tune in, turn on, and quad out.

A quad is the four points that describe the corners of a sprite on the stage in Director 7. It's a sprite property that can be accessed and modified. To see what the quad property looks like, I executed the following code in the message window with a sprite in channel 1 (I'll be using the new Director 7 Lingo dot syntax throughout this article):

```
put sprite(1).quad
— [point(200.0000, 150.0000), point(440.0000,150.0000),
point(440.0000, 330.0000), point(200.0000, 330.0000)]
```

Notice that it's a linear list made up of four elements. Each element is a point (h,v) representing the horizontal and vertical position of a corner of the sprite on the stage. Values are floating point (decimals) to allow for accuracy when manipulating the property. The points are organized in the following order: upper left, upper right, lower right, and lower left.

The power of quads is in modifying the values in the quad "point list" to create perspective and distortion effects. Values in the point list are accessed in the same manner as in any linear list. To determine the value of a particular corner point, use the getAt() function. Here, in the message window, I'll assign a variable I call lowerRightCornerPoint equal to the lower right corner of the sprite in channel 1. I then use a put statement to show that lowerRightCornerPoint contains the desired value:

```
put sprite(1).quad
— [point(200.0000, 150.0000), point(440.0000,150.0000),
point(440.0000, 330.0000), point(200.0000, 330.0000)]
lowerRightCornerPoint = getat(sprite(1).quad,3)
put lowerRightCornerPoint
point(440.0000, 330.0000)
```

Changing the values of the point lists is a bit trickier, since you must set the entire .quad point list at one time. The best way to do this is to set a variable equal to the sprite().quad list. This makes a copy of the point list, which you then modify using the setAt function. When you're finished, set the sprite().quad property equal to your modified variable. Here's how it could look in the message window. I've put the value of sprite(1).quad and quadPointList so you can see how they change throughout the process:

```
— Welcome to Director —
put sprite(1).quad
— [point(36.0000, 38.0000), point(131.0000,38.0000),
point(131.0000, 129.0000), point(36.0000, 129.0000)]
quadPointList = sprite(1).quad
put quadPointList
— [point(36.0000, 38.0000), point(131.0000,38.0000),
point(131.0000, 129.0000), point(36.0000, 129.0000)]
setAt quadPointList, 1, point(999., 999.)
put quadPointList
— [point(999.0000, 999.0000), point(131.0000,38.0000),
point(131.0000, 129.0000), point(36.0000, 129.0000)]
sprite(1).quad = quadPointList
put sprite(1).quad
— [point(999.0000, 999.0000), point(131.0000,38.0000),
point(131.0000, 129.0000), point(36.0000,129.0000)])]
```

When you modify the quad values, Director changes the location of the corner points of the sprite member on the stage, making the image appear as if it were made out of rubber. You can stretch it, shrink it, and even turn it over to see it from behind (as with backwards text). The distorted sprites maintain all their attributes, including ink effects, blend, and behaviors. Even text members remain editable and anti-aliased. Manipulate and distort bitmaps and text in real time to make sophisticated animations with minimal amounts of artwork. Power to the pixels—right on!

Are you experienced?

Two good examples of manipulating quads are supplied by Macromedia in the "About Director" movie (created by Joe Sparks). To access the "About Director" movie, open the Director 7 application. On Macintosh OS machines, select the first item ("About Director...") under the Apple menu at the upper left side of the menu bar. On Windows OS machines, select the last item in the Help pull-down menu ("About Director..."). Once selected, a Movie-In-A-Window will begin playing. Click on the "FUN" button, and when you see the pointing fingers, click on the features bar under the "FUN" button until the "Quads" demo appears. This demo has two parts. The first allows you to manipulate a bitmap by

selecting and grabbing the "magnets" attached to its corners. Flip the image over so you can see the text "Quad Power!" from it's backside. I bet you've never seen a Director bitmap do that before! (Later in this article, I provide an example of how easy it is to create something like this.) While you still have this window open, notice the real-time update of the sprite(32).quadlist at the bottom right-hand side of the window. As the image distorts, you can see how the quad point list values change to reflect the new locations of the images corner points. (See **Figure 1**.)

To get to a place where you no longer need drugs to see the walls breathing, click on the "View Large Quad Animation" button at the lower left side of the window. No, it's not 1969, and someone didn't just spike your coffee. It's 1999 and Director 7 quads. In this example, Joe Sparks created the image of a stone wall with the D7 logo embossed on it (Director 7's code name was Stone). The full image was chopped into an 5x8 grid of equal-size pieces. Each piece was placed in its own channel in the score. The image was then reassembled on the stage using the 40 sprites containing the pieces. Using Lingo, Joe created a series of algorithms and lookup tables to manipulate the quads of the 40 sprites in unison. The result has the beautiful organic feel of the "breathing" wall. The transition to the red hot "lava" wall was achieved by simply swapping in new chopped-up bitmap pieces.

Creating a sophisticated animation like the "breathing wall" is definitely an advanced programming task, but once you create your "breathing image" Lingo engine, you can reuse it again and again. And keep in mind that a movie like this has a very small file size, since only one or two full-screen bitmaps are needed. It's perfect for online delivery.

Breathing walls and floors aren't the only illusions quads can conjure. Imagine 3D objects and worlds you can navigate around and through. I created such a world to act as a navigation interface showing off new Director 7 features for the 415 Productions Web site (www.415p.com / demos / d7l /). Using a gallery metaphor, a visitor can stroll (or run) across a plain containing large monoliths, each announcing a different feature. Clicking on a monolith launches a movie demonstrating the feature. Each "monolith" is a 480x280 pixel bitmap whose quad property is being continuously updated to give the image the correct perspective and create the 3D illusion. Setting quads is very fast, as can be seen from this demo. Navigate behind one of the monoliths containing text and notice you're looking at the "back" of the image, where the text is backwards. The bitmaps themselves are in the JPEG format, so, once again, the file size is very small.

Figure 1. This figure shows the bitmap in the first example turn over by dragging the right side corners across the screen and over the left side of the original image.

Stay at the 415 Productions Director 7 demo movie and navigate to demo 10—"Real Sprite Effects 2" —for a 3D hallucination of another kind. Here, a 3D object has been created by putting together bitmaps to create a solid cube. MouseDown and drag to move the object around. Play with the various combinations of materials, facets, blends, and random buttons to see a few of the amazing effects you can create. (Credit goes to Mark Shepherd for developing the initial code used in both of the 3D examples.)

3D objects and worlds created in Director using quads will undoubtedly be less sophisticated than say, Quake II, and you can't import 3D objects into Director. But with a little imagination, you can create everything from attention-grabbing Web interfaces to next-generation information design. Quads and 3D effects in Director aren't just for gamers—they'll inevitably be used for compelling corporate applications as well.

Try this at home (or in your office)

The trick to creating sprites that distort at your command is knowing how to manipulate the corner points to create the desired effects. You can quickly and easily create interesting effects using the sprite() quad property without having to code a "breathing image" engine like Joe Sparks did or a 3D engine like the one I built for 415 Productions. To get you started, here's an example of a movie where you can grab the corners of a bitmap or text field on stage and drag them wherever you want, distorting the stage image of the object in the process. And you won't have to write a single line of Lingo.

Open up Director 7 and create a new movie (select File / New / Movie). Import a bitmap of a red 1967 Dodge Charger (or any other image) into the internal cast (select File / Import...). Open a text field (select Window / Ttext), and type the name of your favorite Mod Squad character in 48-point Fajita ICG Picante (font style is optional). Set the text to be editable (the editable of member is a Lingo property that can be set to true or false, but it can also be turned on by selecting the Castmember Information button for the Text Castmember / "Options" / "Editable"). Close the Text window, open the Score window, (select Window / Score), and make sure the stage is visible (select Window / Stage).

Select the created text member and the imported bitmap and drag them onto the stage, making sure they're not overlapping. Before we proceed, let's zoom in on the score to make things clearer (select View / Zoom / 1600%).

Now, we're ready to apply the behaviors. Start by opening the Behavior Library Palette (select Window / Library Palette). Select "Java Behaviors" from the Library Palette's upper-left-corner pulldown menu. Select the first behavior "HoldCurrentFrame" from the Java Behaviors and drag it to the score. Place it on the frame script box of frame 1. (Why this commonly used score behavior is placed with the Java Behaviors is beyond my comprehension.) We now have a movie with two channels occupied by a bitmap and text member that will begin playing and loop on frame 1.

Now for the quads part. In the Library Palette, go to the pulldown menu in the upper left corner and select Animation / Interactive. Find the behavior called "Drag Quad Points" and drag it onto the score, placing it on the channel containing the bitmap. When a dialog box appears, dismiss it. Go back to the behaviors palette and drag a copy of the same behavior onto the channel in the score containing the text member. Again, dismiss the dialog box.

At this point, the score of your movie should look something like **Figure 2**. If it doesn't, go back over the instructions again.

You're now ready to run the example (I promised you there would be no Lingo). Close all windows except the stage, rewind the movie (select control / Rewind), and play the movie (select control / Play). Click and drag near the corner of either object on the stage and watch it distort. This behavior is very similar to the example in the "About Director..." movie, sans magnets. Click inside the text member after it's been distorted. Notice that you can still edit the text, even if the text member has been completely flipped over. To quote Jimi Hendrix, "If a 6 turned out to be a 9, I don't mind..."

So how does it work? After determining the corner of the sprite closest to the mouseDown event, the code simply sets the point in the sprite().quad list equal to the mouseDown location. Here are the relevant lines of code from the behavior that are called if the mouse is down:

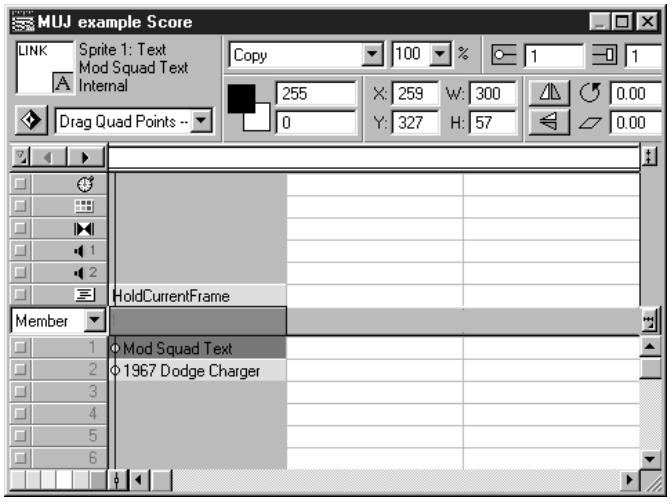

Figure 2. The score with a bitmap and text member in channels 1 and 2. A score behavior keeps the movie on frame1. The sprite quad behavior is on sprites 1 and 2.

```
— get quad from sprite
vQuad = pSprite.quad
— add vector to appropriate position within quadvariable
setAt vQuad, pCurrentQuad, vQuad[pCurrentQuad] + vVector
— set sprite quad to quad variable
pSprite.quad = vQuad
```

First a local variable, vQuad, is set equal to pSprite.quad, so we can manipulate a copy of the quad list. pCurrentQuad is the position in the quad list of the point of closest to the cursor. This is the corner of the sprite that we want to move to the cursor location. vQuad[pCurrentQuad] + vVector determines the location of the cursor. Once the new location of the corner is placed in the correct position in the listvQuad, we set pSprite.quad = vQuad, to reset the corner points of the sprite on stage.

Using these techniques, more advanced programmers can add algorithms or lookup tables, along with the use of multiple sprites, to quickly create robust animations.

The outer limits

Quads aren't without their caveats. Although you can get the .quad of any sprite, you can only manipulate sprites containing bitmap or text members. This means no Quicktime movie cubes or vector art 3D lava lamps.

Setting the quad of a sprite also disables the ability to rotate, skew, and flip the sprite.

Changes made to the .quad list of a sprite do not persist. In other words, when you stop your movie, the sprites will return to their original shape (helpful for recovering severely manipulated images). To restore a sprite to its original quad value using Lingo, set the puppet of the sprite equal to false (sprite(1).puppet = FALSE). ▲

Larry Doyle consults on the full range of Multimedia development. His company, Cyberian Design, provides programming solutions for Web and CD-ROM multimedia projects. He has taught Advanced Director through the San Francisco State Multimedia Studies program and is a regular speaker at the San Francisco Macromedia Users Forum. Past projects include The Rolling Stones Voodoo Lounge CD-ROM, live performance multimedia performance rigs for Peter Gabriel, Shockwave games for SegaSoft, and most recently, the Director 7 feature demos tour for 415 Productions and Macromedia. larry@cyberianDesign.com.

New Text Features . . .

Continued from page 5

screen, you'd have 10 different frames for this screen in the score. In the first frame, select the first editable sprite, and turn on the editable checkbox in the score. In the second frame, select the second editable sprite, and turn on its editable. Repeat ad infinitum, so that in each frame only one of the text entry sprites is actually editable.

To each editable sprite, you'd attach a sprite script in the score that might look something like this:

```
on keyDown
  if the key = TAB then
    if the shiftDown then
      -- go to previous editable sprite
      go to the frame - 1
    else
      -- go to next editable sprite
      go to the frame + 1
  else if the key = RETURN then
    -- if you want the return key to do something
    -- put it here
  else
    -- pass the key pressed on to the editable member
    pass
  end if
end
```

Naturally, the "go frame" commands in this script might look slightly different in the first and last frames if you wanted to have this simulated tabbing loop around. Also, if you wanted to enable the user to set focus via the mouse, you'd probably want to add a marker to each frame and attach a "go to marker__" mouseDown script to each editable sprite.

Troubleshooting editable text problems

If you're having problems when attempting to type editable text, remember that keyboard events get sent to the frontmost window first. If you're in authoring mode (i.e., in Director instead of a projector or a Shockwave movie), make sure the message window, cast window, and so on aren't in front of the stage and therefore intercepting the keystroke. If you have a movie-in-a-window in front of the stage, even if it has no editable sprites itself, it will still need to pass the key event on to the stage with a "tell the stage..." command. If your Shockwave movie isn't responding to key events, it may need to first get the focus in the browser via a mouse-click. (Have your first frame say something like "click here to begin" to ensure this happens.)

Also, remember that with editable text, Director is interacting with operating system elements, and the operating system knows not of things like Director's ink effects and offscreen imaging buffer. For practical purposes, this means that it's difficult to have consistently transparent editable text, and you might have to resign yourself to editable text sprites that have a white background (ink set to "copy" in the score). ▲

Kevin McFarland has worked in the San Francisco Bay area as a freelance Director programmer developing CD-ROMs, corporate presentations, games, kiosks, and prototypes for the past eight years. Kevin@KMC-Interactive.com, www.KMC-Interactive.com.

Technical Editors Ken Durso and Darrel Plant;
Founding Editors Tony Bovè and Cheryl Rhodes;
Publisher Robert Williford;
Vice President/General Manager Connie Austin;
Managing Editor Heidi Frost; **Copy Editor** Farion Grove

Direct all editorial, advertising, or subscription-related questions to Pinnacle Publishing, Inc.:
1-800-788-1900 or 770-565-1763
Fax: 770-565-8232

Pinnacle Publishing, Inc.
PO Box 72255
Marietta, GA 30007-2255

E-mail: **muj@pinpub.com**
Pinnacle Web Site: **http://www.pinpub.com**
Macromedia technical support: 415-252-9080

Macromedia User Journal (ISSN 1065-3929) is published monthly (12 times per year) by Pinnacle Publishing, Inc., 1503 Johnson Ferry Road, Suite 100, Marietta, GA 30062.

POSTMASTER: Send address changes to *Macromedia User Journal*, PO Box 72255, Marietta, GA 30007-2255.