Hacking The Frozen WASTE For Cool Objects
or, Doing Things In, With And To The WASTE Text Engine

Rainer Brockerhoff

Abstract :

Marco Piovanelli's WASTE Text Engine is a popular way to get around the Text Manager's 32K limit; it's used in dozen of great applications, including Netscape, Internet Explorer and many shareware programs.
However, very few applications have explored WASTE's embedded object feature properly, hampered by some limitations in its current implementation (1.3). This paper will show you how to mine WASTE for cool objects, and doing some unexpected things.
<fanfare>
SEE hitherto forbidden lore explained!
SEE the daring explorer battling eldritch bugs!
SEE amazing objects grow, shrink, pulsate and mutate before your very eyes!
SEE multiple QuickTime™ objects playing WHILE U TYPE!
SEE mad Dr. Brockerhoff hacking away at helpless C code!
SEE Marco Piovanelli's hurried disclaimer of all responsibility!!
ONLY here, only at MacHack '99!!!
</fanfare>

Introduction

Some time ago I contracted to port a major Brazilian Portuguese dictionary from Windows to the Mac. Although all text to be displayed was under TextEdit's famous 32K-limit, there was a major snag: depending on conditions, I needed to embed a few icons in the text, standing for literary quotes. Clicking on the icons should show the quotes in a separate window. Unfortunately, TextEdit doesn't allow this sort of thing; embedding a proprietary symbol font in the application would have allowed displaying icons, but copying the text to another application would have copied garbage in the icon's place; and acting on a click would have been completely impossible.

At that point I recalled seeing references to Marco Piovanelli's WASTE (**W**orldScript-**A**ware **S**tyled **T**ext **E**ngine) as a replacement for TextEdit, mostly as a way to get around the 32K limit — which wasn't really needed in my case — but also mentioning its support of embedded graphic objects. WASTE comes on the CodeWarrior CDs and is even used in a version of their SIOUX implementation, so if you program in C, C++ or Pascal you should already have enough material to start programming with WASTE.

WASTE has many things going for it. It's fast, full-featured, and free, and you get full C source code. The author, Marco Piovanelli, only asks for a copy of your application and for appropriate credit in your "About Box". If you use Metrowerk's PowerPlant, Timothy Paustian publishes a CWASTEEdit (soon to become WText) pane class under the same conditions. [Note: WASTE 2.0 should be out by the time you read this — the license now asks for a small fee if you use WASTE in a commercial application]. What's more, there's a WASTE mailing list where Marco and other WASTE sages stand ready to give you support. See the appendix for links and more information.

I assume that you have WASTE 1.3 source available, and that you downloaded CWASTEEdit from the Internet. Most of what I'm going to explain should also work, with little modifications, with WASTE 2.0; at this writing Marco has published an alpha version of 2.0, which already incorporates a few of the additions I'll detail below. A reasonable familiarity with the TextEdit Manager, C/C++, PowerPlant, and Mac programming in general are assumed — this is not for newbies, unfortunately. Batteries not included. Pun alert! Pun alert! Some of the smaller objects discussed here may present a choke hazard. Keep away from programmers under the age of 8. Batteries not included.

A Quick Overflight of the WASTE

WASTE is closely modeled on TextEdit; many calls are similar. In TextEdit, you allocate an **Edit Record** by calling `TENew` (for monostyled text) or `TEStyleNew` (for multistyled text); both return a handle to the edit record, which is a complex structure incorporating all information necessary for a styled text block. All subsequent TextEdit calls are passed the appropriate edit record handle to act upon. There are calls to do copy&paste operations, get and set styles, display parts of the text in a view rectangle, and so forth.

In WASTE, you call `WENew` to allocate a **WE instance**, which is analogous to an edit record. In contrast to TextEdit, the WE instance's structure is opaque; you can access and change nearly all the information through library calls, but you don't have direct access to the data structure. [Note: of course you have all the source code and can expose parts of the structure, or include new routines if you absolutely need to — but normally you shouldn't.] There are also calls for doing copy&paste, get and set styles, display text, drag&drop both to and from WASTE text, and several new calls to accomodate WASTE's extensions. Differently from TextEdit, WASTE uses 32-bit coordinates — since text is limited only by available RAM, the actual rendered size of a text can grow very large — and there are facilities to use inline input for other scripts and languages. Version 2.0 will also have Unicode support, per-paragraph alignment and tabbing, and other innovations. Please refer to the documentation for details.

An important concept in both TextEdit and WASTE is the **style run**. This refers to a continuous block of text which has the exact same font name, size, color, and style throughout.

What we'll discuss mostly here are WASTE's facilities for embedding objects into text. An object is an additional chunk of data which goes to WASTE at any point in the text. Every object has to be of a specific type and you must supply callback routines, or **handlers**, that WASTE calls at the appropriate times. The WASTE routines which calculate the text's layout simply consider each object as a special character (or 'glyph', if you want to be technical) of size such-and-such and the drawing routines call your handler to draw it.

If you copy a TextEdit text block to the clipboard, you'll find that in reality two blocks of data are copied. They're distinguished by what's called a "Scrap Format Type". The actual text is copied as being of type `TEXT`, and the style runs are copied into a separate structure with type `styl`. Copying a WASTE text block to the clipboard also generates `TEXT` and `styl` data, but if there are embedded objects a third data structure is used to hold them, and the associated type is `SOUP`.

Two other interesting WASTE characteristics are that you can get and set various behavior-altering "feature" flags, and there are a number of callback routines which you can install to customize word breaks, click loops, and various other low-level plumbing.

A Bug's View

Every object is represented by a placeholder byte, whose value usually is 0x01, and has an associated style run all for itself. A special field in the object's style run contains a handle to an object of type `WEObjectReference`. The object reference's structure contains the object's attributes: a type tag (4 characters), the object's drawing size (height and width in pixels), and two fields which are specific to each object: a data handle which may point at additional data, and a 32-bit **RefCon**, or reference constant, where you can store any value you wish.

The type tag is used to associate each object with a set of handlers which WASTE calls at several moments : specifically, when an object is created, disposed of, displayed, clicked, and **streamed** to a destination like the clipboard or a file. Simpler objects need not install all handlers, and there are default actions associated with each one. The object's type code is associated with the same code on the clipboard (where it is called a "Scrap Format Type"), so if you copy a color icon to the clipboard — which has the `cicn` scrap format — and paste it into a WASTE text, the "new object handler" associated with the `cicn` type is called.

Now this sounds all very theoretical. Let's look at a particularly simple object to illustrate all this. Suppose you just want to display the word "Square" followed by a blue square, 12 by 12 pixels, like this:

Square 

For this we just need a "draw object handler". The C code in your main routine would look like:

```
#include "WASTE.h"                  // WASTE header file
      LongRect    viewRect;         // these are 32-bit coordinate rectangles
      LongRect    destRect;         // which tell WASTE where to display text
      WEReference weH;        // the WE Instance handle
      const OSType blue='blue';     // our object's type
      Point       size={12, 12};    // our object's size in pixels
                                    // our draw handler's UPP
      WEDrawObjectUPP draw = NewWEDrawObjectProc(ourDrawBlue);
                                    // create empty WE Instance
      weH = WENew(&destRect, &viewRect, 0, &weH);
                                    // tell WASTE about blue's draw handler
      WEInstallObjectHandler(blue, (UniversalProcPtr )weDrawHandler, weH);
                                    // insert 6 characters into text
      WEInsert("Square", 6, nil, nil, weH);
                                    // insert a 'blue' object after that
      WEInsertObject(blue, NewHandle(0), size, weH);
```

Never mind the `nil`'s and `0`'s for now, those are just default parameters. The two `LongRect`'s have to be set to some reasonable value so the text displays OK later. You have to install at least one handler; here we install the draw handler, since without that the object isn't displayed at all. The other handlers are all optional.

Note that we're passing a zero-length Handle to `WEInsertObject`; this is because no extra data are needed for such a simple object. Actually we could pass `nil` here with no ill effects — if running on a recent Mac OS version — but WASTE 1.3 doesn't really treat all ramifications of this, so the orthodox way is preferable.

Let's look at the draw handler for our blue square:

```
pascal OSErr ourDrawBlue(
      const Rect *destRect,         // the destination rectangle
      WEObjectReference objectRef)
{
```

```
	RGBColor blue={0 ,0, 65535};
	RGBBackColor(&blue);		// set background to blue
	EraseRect(&destRect);
	return noErr;			// drawing was successful
}
```

As you see this is very simple: the routine gets passed the object's actual rectangle and must draw inside of that. The `GrafPort` is preset by WASTE. In this case we simply ignore the `objectRef` parameter, since we need no additional information — we just fill the object with blue pixels. But all the handlers get the object reference as parameter, so they can find out more about the object, and if they need to, of its containing WE instance.

A Larger Nugget

Now a blue square is a very simple and inflexible object — the only parameter we can vary from object to object are the height and width passed to `WEInsertObject`. If the RGBColor used by the draw handler were a global variable, we could change the color of all blue squares to another color, but that's all. So let's do a more useful and flexible object: a general picture object. And we now just need to look at the handlers themselves… installing them follows the example given above.

To be able to copy&paste pictures from the clipboard, it's easiest to associate our handlers to the `PICT` type.; that way we can just paste any picture, like this one  inside our text.

We need to store the data somewhere and to display each picture at it's correct size, so we need a "new object handler". This would be:

```
pascal OSErr ourNewPICT(
	Point *defaultObjectSize,	// return the size here
	WEObjectReference objectRef)
{
					// retrieve handle to PICT data
	PicHandle   pict = (PicHandle)WEGetObjectDataHandle(objectRef);
	if (pict) {			// should never be nil, but who knows?
					// calculate width
		(*defaultObjectSize).h = (*pict)->picFrame.right
						- (*pict)->picFrame.left;
					// calculate height
		(*defaultObjectSize).v = (*pict)->picFrame.bottom
						- (*pict)->picFrame.top;
	}
	return noErr;			// picture created OK
}
```

This is also easy. When the new handler is called, the `PICT` data are already stored in the object's reference structure and are retrieved by `WEGetObjectDataHandle`. Since we already know this is a handle to a `PICT`, we just calculate the picture's height and width and store them in the `defaultObjectSize` parameter. This parameter alwasy comes preset to `{32,32}`; if we pass `{0,0}` as the object's dimensions in the `WEInsertObject` call — as is always done by a paste operation — the size calculated by the new handler is used. This allows us to override the object's size if we call `WEInsertObject` explicitly.

It's nearly always advisable to return `noErr` from any object handler; if you return any error value it gets propagated upward and have untoward consequences if the error gets handed to PowerPlant or some other library. We'll see an exception to this later.

When the picture object is destroyed (as might happen if we type over it, or do a cut), WASTE calls a "dispose object handler". Strictly speaking we don't need a dispose handler here, since the default action if it's not present is to simply dispose of the object data handle. But it's a good opportunity to show a simple dispose handler:

```
pascal OSErr ourDisposePICT(
      WEObjectReference objectRef)
{
                                    // retrieve handle to PICT data
      PicHandle   pict = (PicHandle)WEGetObjectDataHandle(objectRef);
      if (pict) {             // should never be nil, but who knows?
            KillPicture(thePic);    // dispose
      }
      return noErr;                 // picture destroyed OK
}
```

Let's now see how the actual drawing of the picture object should be done:

```
pascal OSErr ourDrawPICT(
      Rect *destRect,               // the destination rectangle
      WEObjectReference objectRef)
{
                                    // retrieve handle to PICT data
      PicHandle   pict = (PicHandle)WEGetObjectDataHandle(objectRef);
      if (pict) {             // should never be nil, but who knows?
            DrawPicture(pict, destRect);
      }
      return noErr;                 // picture displayed OK
}
```

Nothing could be more straightforward. Since the `destRect`'s dimensions are usually those we calculated in the new handler, `DrawPicture` will draw the picture at the normal size. WASTE handles the actual text layout just as if the picture were a (usually somewhat large) character in a special font.

Handling our Treasures

Let's take the picture object as an excuse to show the other two handlers. Oddly enough, the "click object handler" is called by WASTE whenever the object is clicked. There are some restrictions to the usual way WASTE handles clicking — you have to click once on the object to select it, then again to call the click handler — and if you react to single clicks, you won't let the user start a drag action. We'll see later how to get around this. For the moment, suppose we want to beep when the picture is double-clicked (which actually means triple-clicked, if it's not already selected).

```
pascal Boolean ourClickPICT(
      Point hitPt,                  // the click location
      short modifiers,        // the click's modifier keys
                                    // + bit 0 set if double-click
      long clickTime,         // time when click happened
      WEObjectReference objectRef)
{
      if (modifiers & 0x0001) {     // check if double-click
            SysBeep(1);       // then beep
            return true;            // we handled the click
```

```
	}
	return false;			// we didn't handle the click
}
```

We don't need the click location and time here, and make sure it was a double-click. Notice that this handler returns a Boolean instead of an error code. Return true to mean that you've handled the click, or false to tell WASTE to handle the click itself — we'll explain why further ahead.

Now let's say we have a picture which we want to display but we don't want the user copying or dragging this picture to another application. So we'll code a "streaming object handler" to return a text message instead:

```
pascal OSErr ourStreamPICT(
	SInt16 destKind,		// where we're streaming to
	FlavorType *theType,		// return data flavor
	Handle putDataHere,		// return streamed data here
	WEObjectReference objectRef)
{
	const UInt8* message="\pFooled you!!";
				// check destination
	switch (destKind) {
	case weToDrag:			// dragging the object
	case weToScrap:			// copying to clipboard
				// test if WASTE wants data
		if (putDataHandle) {
				// set output handle size
			SetHandleSize(putDataHere, *message);
				// be sure this worked
			if (MemError()==noErr) {
				// copy the message
				BlockMoveData(message, *putDataHere, *message+1);
			}
				// tell WASTE the output type
			*theType = 'TEXT';
			return noErr;	// streaming was successful
		};
		break;
	case weToSoup:			// storing internally
				// default to normal action
		break;
	};
	return weNotHandledErr; // tell WASTE to do it
}
```

This is a more complicated handler. The `destKind` parameter says whether we're streaming to a drag destination, to the clipboard, or to a `SOUP` — this third case usually happens when you are saving the text to disk. On entry, `theType` contains the object's type, but you can change this to another recognized type, as we're doing here. The third parameter, `putDataHere`, can be either `nil` or a zero-length handle on entry. You use this to return the data you want to be streamed to the particular destination for that call. But if it is `nil` this means WASTE is just asking you which data type you'll return later.

Here's a case where the return value can vary. If you return `noErr`, you're telling WASTE to use the data and data type you're returning in the other parameters. If you return `weNotHandledErr`, this means that WASTE should just use the object's data as usual — as if there were no installed handler.

So our streaming handler allows the user to save the file containing our special picture to disk, but if he tries to drag or copy the picture to another place, he gets a cheeky text message instead. In practice, the streaming handler can be used to massage complicated objects which are stored in multiple handles — icon families, for instance. You need to "flatten" the data into a single handle and pass that out. Your new object handler gets the flattened data back later, and must "unflatten" it to restore the original structure, so it can be used.

The Sound of One Hand Clicking

So now you've met all five kinds of object handlers. With this information you can write simple objects that display fixed graphics like pictures and icons, and that can be clicked. Indeed, by following the links in the Appendix you can find source code to implement several such objects; the standard WASTE distribution includes code (written by Michael F. Kamprath with help from John C. Daub) that is very similar to our picture object. It also includes a sound object that displays a fixed icon and plays a sound when double-clicked, and a HFS object that encodes a file location and opens that file when you double-click it. Unfortunately that's about it — I could find no other published object types.

Now we're going to get serious about clicking on objects. First of all, it's clear (with apologies to Marco Piovanelli) that the current WASTE click routine, called `WEClick`, is somewhat inefficient in its object handling. As I said above, there are two main limitations: the fact that you have to select an object first, and then single-or double-click on it; and if you single-click on it and handle the single-click (which now is actually a double-click) in your click handler, the object becomes undraggable. And, depending on your timing or on the prior selection state of an object, you have to either double- or triple-click to have your handler called.

A secondary issue is the way that clicking on an object is handled: to select it, you must click in the middle two quarters of the object' rectangle; if you click on the edge quarters WASTE treats the object like a character glyph and moves the insertion point to just before or after the object, instead of calling you click handler.

So I've written a new `WEClick` routine which works better. The main idea is to have several kinds of objects (as related to click handling), which work as follows:
• A "dumb" object's click handler always returns `false`. It's handled as if it were a glyph for selection and dragging purposes; the only real difference between a character and a "dumb" object is that you can select the object by clicking on its middle.
• A "smart" object always return `true`. It must be selected by some means other than clicking in its middle — like, for instance, dragging the cursor over it — or it may elect to return `false` (play "dumb") if clicked with some modifier key held down.
• A "legacy" object returns `false` on a single-click but returns `true` on a double-click. These should continue working except that "legacy" objects which return `true` on a double-click will not alter the selection; if you want "legacy" objects to work exactly as before just modify their handlers to always return `false`. Our example `PICT` object, earlier, behaves as a "legacy" object.

Any click within an object is analyzed as to what its effects would be for a glyph. If the click would cause the object to be selected, or the insertion point to be set immediately before or after the object, the click handler is called… this means that all clicks within the object are handled, except for shift-clicks which are extending the selection range.

If that object (and only that object) would normally be selected after the click, bit 1 of the "modifiers" argument to the handler is set. This may be useful in special cases.

If the handler returns `true`, it handled the click completely, and `WEClick` exits. This means that the selection is *not* altered. This is useful for objects that implement, for instance, QuickTime movies.

If the handler returns `false`, it may have done something, but for all intents WASTE will handle the click further (as if the object were a glyph). This extends to double-clicking; meaning that an object will get a double-click call (bit 0 of "modifiers" set) only if its handler returned `false` to the first click. "Smart" objects will get both clicks of a double-click as two single-clicks, and must therefore do their own click timing analysis to see if they've been double-clicked.

Notice that if you select a "smart" object it can then be dragged normally (by dragging from its middle) but will get other clicks normally. This may admittedly be somewhat confusing. Another side-effect is that if you have a smart object at offset zero, you can't normally set the insertion point before it by just clicking near its left edge — unless you also call `WEClick` for clicks in the left margin of the WASTE instance. You can do this using the "margins" feature of CWASTEEdit, for instance.

In general, all objects we'll discuss after this point will be "smart" objects (unless they don't have a click handler at all), and we'll suppose that you installed the patched `WEClick` routine. You can find it in Appendix A; it's too long to insert here in the main text. Just paste it into the WASTE file "WEMouse.c".

Speak Softly and Carry a Variable-Size Stick

The original intention of WASTE's object handler API was to isolate the object's implementation from its surroundings, and presupposed a static object which doesn't change throughout its lifetime. The `objectRef` parameter is an opaque handle and usually you just call `WEGetObjectDataHandle(objectRef)` to get at the data handle. There are a few similar calls: `WEGetObjectType` to get the object's type, `WEGetObjectSize` to get its size, `WEGetObjectRefCon` and `WESetObjectRefCon` to set and get its RefCon value, and `WEGetObjectOwner` to obtain its "owning" WE instance handle. As we saw, these humdrum objects are not very exciting. Let's begin by hacking WASTE to include a new call to alter the size of an existing object — `WESetObjectSize`. This call, in fact, has already been included in WASTE 2.0, so I'm giving Marco's implementation of it here, or rather, in Appendix B, along with another routine it needs: `WEGetObjectOffset`, which (surprise!) returns the offset of the current object's marker byte.

Now we can do new things with our old `PICT` object's click handler. Let's rewrite it as follows:

```
pascal Boolean ourClickPICT(
      Point hitPt,                  // the click location
      short modifiers,        // the click's modifier keys
                                    // + bit 0 set if double-click
                                    // + bit 1 set if click will select
      long clickTime,         // time when click happened
      WEObjectReference objectRef)
{
                                    // get current size
      Point size = WEGetObjectSize(objectRef);
                                    // option-click zooms in
      if (modifiers & optionKey) {
            size.h <<= 1;           // double width
            size.v <<= 1;           // double height
            WESetObjectSize(objectRef, size);
      } else
                                    // control-click zooms out
      if (modifiers & controlKey) {
```

```
            size.h >>= 1;           // halve width
            size.v >>= 1;           // halve height
            WESetObjectSize(objectRef, size);
      }
      return true;                  // always handle the click
}
```

As you can see by the last line, this is a “smart” picture object. It always handles the click, so it never gets double-clicks under our new scheme. Option-clicking always doubles the size of the object, and control-clicking halves it. [Note: this is a little crude, as zooming out from the original size will lose significant bits in the `size` fields and zooming in again will never exactly restore the original size; a real-world implementation might shift the lost bits into the objects RefCon, and shift them out again later].

Today my Neighbor Object, Tomorrow the World

The original APIs were never intended to have an object find out about its surroundings or, heaven forfend, affect other objects. Yet, that’s exactly what we’re going to do now: a “remote-control zoom” object which controls the size of the picture that follows it. Let’s write a `Cntl` object to be the controller, and keep using our previous (non-clickable) `PICT` object as the controllee.

For simplicity’s sake I’m going to do the controller as an 8-bit color icon from an icon family (in other words, an `icl8` resource). Since such an icon is 32x32 pixels, and we don’t need to store the icon data, we don’t need a new handler, nor a dispose handler.

The draw handler just draws the icon: 


```
pascal OSErr ourDrawCntl(
      Rect *destRect,               // the destination rectangle
      WEObjectReference objectRef)
{
                                    // suppose resource's ID is 128
      PlotIconID(destRect, kAlignNone, kTransformNone, 128);
      return noErr;                 // icon displayed OK
}
```

All the magic incantations are done in the click handler:

```
pascal Boolean ourClickCntl(
      Point hitPt,                  // the click location
      short modifiers,        // the click's modifier keys
                                    // + bit 0 set if double-click
                                    // + bit 1 set if click will select
      long clickTime,         // time when click happened
      WEObjectReference objectRef)
{
      Rect frame;
      WEObjectReference next = nil;
      Point size;
                                    // get our WE instance handle
      WEReference weH = WEGetObjectOwner(objectRef);
                                    // get the icon's frame
      WEGetObjectFrame(objectRef, &frame);
                                    // find our own offset in the text
      SInt32 offset = WEGetObjectOffset(objectRef);
```

```
                              // loop over the next objects
      while ((offset = WEFindNextObject(offset, &next, weH)) >= 0) {
                              // make sure it's a PICT
            if (WEGetObjectType(next) == 'PICT') {
                              // get its current size
                  size = WEGetObjectSize(next);
                              // clicking top half zooms in
                  if (hitPt.v < (frame.top+frame.bottom)/2) {
                        size.h <<= 1;           // double width
                        size.v <<= 1;           // double height
                        WESetObjectSize(next, size);
                  } else {          // clicking bottom half zooms out
                        size.h >>= 1;           // halve width
                        size.v >>= 1;           // halve height
                        WESetObjectSize(next, size);
                  }
                  break;            // stop looping
            }
      }
      return true;                  // always handle the click
}
```

To find out which part of the icon was clicked, we need information about the icon's actual on-screen coordinates, which aren't available in standard WASTE. So I wrote the `WEGetObjectFrame` routine, which is in Appendix C. This routine returns the object's rectangle in the same coordinates used by the `hitPt` parameter.

The click handler loops over all objects which follow the remote control. Successive calls to `WEFindNextObject` do this easily, returning a new object reference everytime. Once a `PICT` is found, we get its size and then grow or shrink it, depending on where the control was clicked.

The Magic Transporter

Since we're now free to mess around with companion objects, let's do a hypertext link object and its counterpart, the anchor. For simplicity's sake, we'll make a single `Link` object fill both roles, and limit the link's size to 255 characters.

Our hypertext links are always displayed in blue underlined 10-point Geneva, and our main routine has already obtained a `FMOutput` structure with the metrics for that font size. Anchors are simply displayed as a blank pixel. We also need a new data structure for our `Link` object:

```
typedef struct linkData {
      UInt16      ID;               // the link's identification number
      UInt8       name[];           // leave name empty for an anchor
} linkData;
extern FMOutput geneva10;     // metrics for Geneva 10
```

This structure — or rather a handle to it — must be passed as the second argument to `WEInsertObject` when you're creating a new link or anchor. Since `Link` isn't a standard scrap format type, we can't create one by pasting; you'll want to have a menu command called "Insert Link", for instance, which opens a dialog box where you fill in the link name and ID, and calls `WEInsertObject` appropriately. I'll leave this as an exercise for the student.

The new handler is quite simple, and we don't need a dispose handler:

```
pascal OSErr ourNewLink(
```

```
      Point *defaultObjectSize,     // return the size here
      WEObjectReference objectRef)
{
                                    // retrieve handle to link data
      linkData**  link = (linkData**)WEGetObjectDataHandle(objectRef);
      if (link) {             // should never be nil, but who knows?
                                    // check if it's a link or an anchor
            if ((*link)->name[0]) {
                                    // it's a link
                  TextFont(kFontIDGeneva);
                  TextSize(10);
                  TextFace(underLine);
                                    // measure name's dimensions
                  (*defaultObjectSize).h = TextWidth((*link)->name,
                                          1, (*link)->name[0]);
                  (*defaultObjectSize).v = geneva10.ascent+geneva10.descent;
            } else {
                                    // it's an anchor, so set to {1,1}
                  (*defaultObjectSize).h = 1;
                  (*defaultObjectSize).v = 1;
            }
      }
      return noErr;                 // link created OK
}
```

As you can see, the name is a null string for an anchor, and we calculate the dimensions accordingly to the contents of the name. A curious side-effect is that a hypertext link seems to be text, but in reality is an indivisible object; it can't be edited directly.

The draw handler is very similar in structure:

```
pascal OSErr ourDrawLink(
      Rect *destRect,               // the destination rectangle
      WEObjectReference objectRef)
{
                                    // retrieve handle to link data
      linkData**  link = (linkData**)WEGetObjectDataHandle(objectRef);
      if (link) {             // should never be nil, but who knows?
                                    // check if it's a link or an anchor
            if ((*link)->name[0]) {
                                    // it's a link, set text style
                  RGBColor blue={0 ,0, 65535};
                  RGBForeColor(&blue);
                  TextFont(kFontIDGeneva);
                  TextSize(10);
                  TextFace(underLine);
                                    // position pen
                  MoveTo(destRect->left, destRect->top+geneva10.ascent);
                                    // draw the link name
                  DrawText((*link)->name, 1, (*link)->name[0]);
            } else {
                                    // it's an anchor, so just erase it
                  RGBColor white={65535, 65535, 65535};
                  RGBBackColor(&white);
                  EraseRect(&destRect);
            }
      }
      return noErr;                 // link created OK
}
```

Now comes the hard part. The click handler has to find the first anchor with the same ID number and scroll the text to that point. Let's also give some standard UI feedback, inverting the link while the mouse is down, but jump to the anchor only if the mouse is released while still inside the link.

```
pascal Boolean ourClickLink(
      Point hitPt,                  // the click location
      short modifiers,        // the click's modifier keys
                                    // + bit 0 set if double-click
                                    // + bit 1 set if click will select
      long clickTime,         // time when click happened
      WEObjectReference objectRef)
{
      linkData**  link = (linkData**)WEGetObjectDataHandle(objectRef);
      if (link) {             // should never be nil, but who knows?
                                    // check if it's a link or an anchor
            if ((*link)->name[0]) {
                                    // it's a link, wait for mouse-up
                  Rect frame;
                  Boolean in=true, inv=false, old=false;
                                    // get our frame rect
                  WEGetObjectFrame(objectRef, &frame);
                                    // loop while button is down
                  while (StillDown()) {
                                    // get new mouse location
                        GetMouse(&mouseLoc);
                                    // check if still in frame
                        in =  PtInRect(mouseLoc, &frame);
                                    // invert link if situation changed
                        if (in!=old) {
                              InvertRect(&frame);
                              inv = !inv;
                              old = in;
                        }
                  }
                                    // button has gone up
                  if (inv) {
                                    // return to normal
                        InvertRect(&frame);
                  }
                                    // check where the mouse is now
                  GetMouse(&mouseLoc);
                  if (PtInRect(mouseLoc, &frame)) {
                                    // mouse released inside link
                        WEObjectReference next = nil;
                                    // get our WE instance handle
                        WEReference weH = WEGetObjectOwner(objectRef);
                        linkData**  anchor;
                        SInt32 offset = kInvalidOffset;
                                    // loop over objects from start
                        while ((offset = WEFindNextObject(
                                    offset, &next, weH)) >= 0) {
                                    // make sure it's a link
                              if (WEGetObjectType(next) == 'Link') {
                                    // get the link data
                                    anchor = (linkData**)
                                                WEGetObjectDataHandle(next);
                                    // make sure it's an anchor
                                    if ((anchor*)->name[0]==0) {
```

```
                                    // check the ID number
                                          if ((anchor*)->ID==(link*)->ID) {
                                    // found the anchor, scroll to it
                                                scrollToOffset(offset, weH);
                                    // and stop here
                                                return true;
                                          }
                                    }
                              }
                        }
                                    // anchor not found, beep
                        SysBeep(1);
                  }
            }
      }
      return true;                  // always handle the click
}
```

For clarity's sake, the `scrollToOffset` part was made into a separate routine. I show it here just for completeness, since it has nothing to do with our object handlers; it tries to scroll the text so that the anchor's line is at the top.

```
void scrollToOffset(
      SInt32 offset,
      WEReference weH)
{
      LongRect viewRect, destRect;
      LongPt pt;
      Sint32 disp;
      WESetSelection(offset+1, offset+1, weH);
      WEGetViewRect(&viewRect, weH);
      WEGetDestRect(&destRect, weH);
      WEGetPoint(offset, leftCaret, &pt, nil, weH);
      disp = destRect.bottom-viewRect.bottom;
      if (disp > pt.v) {
            disp = pt.v;
      }
      if (disp < (destRect.top-viewRect.top)) {
            disp = destRect.top-viewRect.top;
      }
      if (disp) {
            WEOffsetLongRect(&destRect, 0, disp);
            WESetDestRect(&destRect, weH);
            WEUpdate(nil, weH);
      }
}
```

Ignore the Object Behind the Curtain

Now we're going to tackle something more complex: a QuickTime movie object. There are two main issues here: one is that the embedded movie's frame has to be always in step with the movie object's frame, so you can scroll the text while the movie is playing; and the second is that, well, the movie might be playing at any time and needs a steady stream of your application's attention — in the form of events.

Normally we would use the standard `moov` type for our object, so you would — in theory, at least — be allowed to paste in a movie from the clipboard, or drag it in from the Finder

Unfortunately, I'll have to skip over an important part of the process here: properly preparing a QuickTime movie for the `WEInsertObject` call. The catch is that when you simply copy&paste, or drag, a movie, this passes a movie reference rather than a self-contained movie; that is, the resulting object simply points back at the original file which must remain in that place for the object to work. While this may be desirable for some applications and very large movies, it means the resulting text block isn't self-contained; you need to keep the original movie at its original location.

For my application I solved the drag problem with a rather complex routine which reads in the movie file, writes a "flattened" copy to a temporary file, reads it back into memory, and fudges the references to refer to the copy in memory. QuickTime 3.0 and up has facilities to do this in RAM, but it's still a tricky process, and too long to detail here.

So we'll use a non-standard `Moov` type instead, and read in a fixed movie file from disk. In your main program you'll need to have as globals:

```
typedef struct movieData {
      Movie             movie;
      MovieController   player;
      RgnHandle         region;
      Rect              frame;
      UInt32            flags;
      LView*            view;
      Boolean           active;
      Boolean           empty;
} movieData;
WEObjectRef movieObject=nil;
```

Notice that this allows for only one movie object— in real life, there would be a list of active movie objects.

And later at a convenient point, you will do:

```
      movieData** mH = NewHandleClear(sizeof(movieData));
      FSSpec theSpec={0,0,"\pMy Movie"};
      Point size={0,0};
      SInt16 resRefNum, resID;
      Str255 movieName;
      Boolean chg;
                  // open the movie file (should check for errors!)
      OpenMovieFile(&theSpec, &resRefNum, fsRdPerm);
                  // make a movie reference (should check for errors!)
      NewMovieFromFile((*mH)->movie, resRefNum, &resId, theMovie, 0, &chg);
                  // store the view the WE instance belongs to
      (*mH)->view = theWASTEview;
                  // and supposing weH is already set up
      WEInsertObject('Moov', (Handle)mH, size, weH);
```

Here I'm assuming you're using PowerPlant and are drawing the WASTE text into some LView descendant. This makes things easier later.

The new handler for that would then be:

```
pascal OSErr ourNewMoov(
      Point *defaultObjectSize,     // return the size here
      WEObjectReference objectRef)
{
      Rect box;
      GrafPtr gp;
```

```
                                    // get the movie data structure
      movieData** data = (movieData**)WEGetObjectDataHandle(objectRef);
      if (data) {             // should never be nil, but who knows?
                                    // make sure movie's box is
                                    // aligned at top left
            GetMovieBox((*data)->movie, &box);
            OffsetRect(&box, -box.left, -box.top);
            SetMovieBox((*data)->movie, &box);
                                    // make a movie controller
            (*data)->player = NewMovieController((*data)->movie,
                  &box, mcTopLeftMovie);
                                    // standard position
            MCPositionController((*data)->player, &box, nil,
                  mcTopLeftMovie|mcPositionDontInvalidate);
                                    // set controller to be visible
            MCSetVisible((*data)->player, true);
                                    // get controller bounds
            MCGetControllerBoundsRect((*data)->player, &(*data)->frame);
                                    // this will be WASTE's GrafPort
            GetPort(&gp);
                                    // clip movie to WASTE's clipRgn
            MCSetClip((*data)->player, gp->clipRgn, gp->clipRgn);
                                    // keep inactive for now
            MCDoAction((*data)->player, mcActionDeactivate, nil);
            (*data)->active = false;
                                    // empty means there's no controller
            (*data)->empty = EqualRect(&box, &(*data)->frame);
                                    // this region will be filled later
            (*data)->region = NewRgn();
                                    // get the movie size in pixels
            *defaultObjectSize = botRight((*data)->frame);
                                    // make sure it's non-zero
            if (defaultObjectSize->v<1) {
                  defaultObjectSize->v = 1;
            }
            if (defaultObjectSize->h<1) {
                  defaultObjectSize->h = 1;
            }
                                    // store our reference in a global
            movieObject = objectRef;
      }
      return noErr;                 // movie created OK
}
```

This actually rather straightforward, although we need to set up a lot of stuff for later use. And we need a dispose handler, which is simply:

```
pascal OSErr ourDisposeMoov(
      WEObjectReference objectRef)
{
                                    // get the movie data structure
      movieData** data = (movieData**)WEGetObjectDataHandle(objectRef);
      if (data) {             // should never be nil, but who knows?
                                    // stop the movie in case it's playing
            StopMovie((*data)->movie);
                                    // clear the global pointer
            movieObject = nil;
                                    // dispose of the region
            if ((*data)->region) {
```

```
                DisposeRgn((*data)->region);
        }
                                // dispose of the movie controller
        DisposeMovieController((*data)->player);
                                // dispose of the movie itself
        DisposeMovie((*data)->movie);
                                // dispose of the movie data structure
        DisposeHandle((Handle)data);
    }
    return noErr;               // movie destroyed OK
}
```

Now we need a draw handler, too:

```
pascal OSErr ourDrawMoov(
    Rect *destRect,             // the destination rectangle
    WEObjectReference objectRef)
{
    GrafPtr gp;
                                // get the movie data structure
    movieData** data = (movieData**)WEGetObjectDataHandle(objectRef);
    if (data) {             // should never be nil, but who knows?
                                // if the movie is inactive or has been
                                // scrolled elsewhere, update location
        if (!(*data)->active || !EqualRect(&(*data)->frame, destRect)) {
                                // remember this location
            (*data)->frame = *destRect;
                                // set either movie or controller's location
            if ((*data)->empty) {
                SetMovieBox((*data)->movie, &(*data)->frame);
            } else {
                MCSetControllerBoundsRect((*data)->player,
                    &(*data)->frame);
            };
        };
                                // this will be WASTE's GrafPort
        GetPort(&gp);
                                // get the PowerPlant view's frame
        if ((*data)->view) {
            ((*data)->view)->CalcLocalFrameRect(frame);
                                // make it into a region
            RectRgn((*data)->region,&frame);
                                // clip movie to the view
            MCSetClip((*data)->player,(*data)->region,(*data)->region);
        };
                                // have QuickTime redraw the movie
        MCDoAction((*data)->player,mcActionDraw,gp);
                                // let's start the movie automatically
        if (!(*data)->active) {
            StartMovie((*data)->movie);
            (*data)->active = true;
        };
    }
    return noErr;               // movie drawn OK
}
```

And we need a click handler:

```
pascal Boolean ourClickLink(
    Point hitPt,                // the click location
```

```
	short modifiers,		// the click's modifier keys
					// + bit 0 set if double-click
					// + bit 1 set if click will select
	long clickTime,		// time when click happened
	WEObjectReference objectRef)
{
	GrafPtr gp;
					// fake up a mouse down event
	EventRecord er={mouseDown,	nil, clickTime, hitPt, modifiers&(~3)};
					// get the movie data structure
	movieData** data = (movieData**)WEGetObjectDataHandle(objectRef);
	if (data) {			// should never be nil, but who knows?
					// make sure we have view
		if ((*data)->view) {
					// convert mouse to global
			((*data)->view)->LocalToPortPoint(er.where);
			((*data)->view)->PortToGlobalPoint(er.where);
					// this will be WASTE's GrafPort
			GetPort(&gp);
					// set event message to GrafPort
			er.message = (UInt32)gp;
					// have QuickTime handle the click
			MCIsPlayerEvent((*data)->player,&er);
		}
	}
	return true;			// always handle the click
}
```

The click handler simply reconstitutes the original mouse-down event and passes it to QuickTime. That should be all there is to it, right?

No! QuickTime need constant attention. In your main event loop, you need to call a routine that gives QuickTime a chance to see the event first. Since we're using PowerPlant here, this is easy; just override your application's `ProcessNextEvent` routine, like this:

```
void ourApplication::ProcessNextEvent()
{
	EventRecord macEvent;
	if (IsOnDuty()) {
		OSEventAvail(0, &macEvent);
		AdjustCursor(macEvent);
	}
	SetUpdateCommandStatus(false);
	Boolean gotEvent = WaitNextEvent(everyEvent, &macEvent, mSleepTime,
							mMouseRgn);
	if (!wasQuickTimeEvent((&macEvent)) {
		if (LAttachable::ExecuteAttachments(msg_Event, &macEvent)) {
			if (gotEvent) {
				DispatchEvent(macEvent);
			} else {
				UseIdleTime(macEvent);
			}
		}
	}
	LPeriodical::DevoteTimeToRepeaters(macEvent);
	if (IsOnDuty() && GetUpdateCommandStatus()) {
		UpdateMenus();
	}
}
```

Here we just inserted a line to call `wasQuickTimeEvent` before letting PowerPlant handle the event, if QuickTime didn't want to. This routine is as follows:

```
Boolean wasQuickTimeEvent(const EventRecord* inMacEvent) {
      Rect frame;
      GrafPtr gp;
      SInt32 flags;
      Boolean result=false,handled;
                              // make a copy of the event
      EventRecord evt=*inMacEvent;
      switch (evt.what)       // discard some event types outright
      case mouseDown:
      case activateEvt:
      case diskEvt:
      case kHighLevelEvent:
            return false;
      default:
                              // get the movie data structure
            movieData** data = (movieData*)
                              WEGetObjectDataHandle(movieObject);
            if (data) {       // should never be nil, but who knows?
                              // get the movie's frame
                  WEGetObjectFrame(movieObject, &frame);
                              // get and focus the view
                  if ((*data)->view && ((*data)->view)->FocusDraw()) {
                              // if the movie is inactive or has been
                              // scrolled elsewhere, update location
                        if (!EqualRect(&frame,&(*data)->frame)) {
                              // remember this location
                              (*data)->frame = frame;
                              // set either movie or controller's location
                              if ((*data)->empty) {
                                    SetMovieBox((*data)->movie, &(*data)->frame);
                              } else {
                                    MCSetControllerBoundsRect((*data)->player,
                                          &(*data)->frame);
                              }
                        }
                              // get the view's frame region
                        ((*data)->view)->CalcLocalFrameRect(frame);
                        RectRgn((*data)->region, &frame);
                              // clip movie to the view
                        MCSetClip((*data)->player, (*data)->region,
                              (*data)->region);
                              // get info from controller
                        MCGetControllerInfo((*data)->player, &flags);
                              // redraw movie if it's not playing
                        if (!(flags & mcInfoIsPlaying)) {
                              GetPort(&gp);
                              MCDoAction((*data)->player, mcActionDraw, gp);
                        }
                              // see if QuickTime handled the event
                        result = MCIsPlayerEvent((*data)->player, &evt)!=0;
                              // let PowerPlant handle these events too
                        if ((evt.what==nullEvent) || (evt.what==updateEvt)) {
                              result = false;
                        }
                  }
```

```
            }
        }
        return result;
}
```

This repeats some of the actions of the draw handler to ensure that the movie is always redrawn accurately. As you can see, certain events are always handled by PowerPlant only, others by QuickTime, others by both. Here too, in real life, instead of just handling a single movie object you would need to iterate over a list of active movies.

A Voracious Beast

Now that we've found ways to give periodical attention to our objects, they can be active at any time and almost seem alive. And their reach can extend to the entire text. Here's a rare beast that I found on my last trip into the WASTE: a Hex caterpillar. Click on its head, and it will eat forwards into your text and excrete each character into a messier form (as usually happens), as its hex equivalent. Fortunately you can click on the caterpillar's tail and persuade it to crawl backwards to reverse the process…
Let's suppose you have a nice picture of a caterpillar, like this: . After you're done snickering at the anatomical details, you can simply reuse the new, drawing and dispose handlers from the `PICT` object described above. As so often happens, all the dirty work will be done by the click handler:

```
pascal Boolean ourClickCaterpillar(
        Point hitPt,                    // the click location
        short modifiers,          // the click's modifier keys
                                        // + bit 0 set if double-click
                                        // + bit 1 set if click will select
        long clickTime,           // time when click happened
        WEObjectReference objectRef)
{
        Rect frame;
        WEObjectReference next = nil;
        Point size;
        SInt16 chr,chr2;
                                        // get our WE instance handle
        WEReference weH = WEGetObjectOwner(objectRef);
                                        // get the pictures's frame
        WEGetObjectFrame(objectRef, &frame);
                                        // find our own offset in the text
        SInt32 offset = WEGetObjectOffset(objectRef);
                                        // check where user clicked
        if (hitPt.v > (frame.left+frame.right)/2) {
                                        // clicked in the head half
                                        // so get the char that follows
              chr = WEGetChar(offset+1, weH);
              if (chr>0) {                    // check that there is one
                                        // select the char
                    WESetSelection(offset+1, offset_2, weH);
                    WEDelete(weH);    // and delete it
                                        // set insertion point
                    WESetSelection(offset, offset, weH);
                                        // insert hex equivalent of char
                    WEKey(HexDigitToChar(chr>>4), 0, weH);
                    WEKey(HexDigitToChar(chr&0x0f), 0, weH);
```

```
            }
        } else {                        // clicked in the tail half
                                        // so get 2 chars before
            chr = WEGetChar(offset-1, weH);
            chr2 = WEGetChar(offset-2, weH);
                                        // check they exist
            if ((chr>0)&&(chr2>0)) {
                                        // convert both to hex digits
                chr = CharToHexDigit(chr);
                chr2 = CharToHexDigit(chr2);
                                        // check for success
                if ((chr>=0)&&(chr2>=0)) {
                                        // set insertion point
                    WESetSelection(offset+1, offset+1, weH);
                                        // insert char equivalent of hex
                    WEKey((chr2<<4)+chr, 0, weH);
                };
            }
        }
        WEUpdate(nil, weH);             // redraw
        return true;                    // always handle the click
}
```

To shorten our example here, I didn't bother with saving and restoring the selection, and delegated the Hex to ASCII and back conversions to external routines `HexDigitToChar` and `CharToHexDigit`. Also, modifying the example to do some sort of animation of the caterpillar's gobbling, chewing and umhhh… excreting… together with appropriate sound effects, is left as an exercise for the reader.

Space Precludes

According to "The Computer Contradictionary" (an invaluable reference work by the immortal Stan Kelly-Bootle), the above heading indicates "ignorance cloaked in a long-winded claim that the author's word-allocation has been exceeded". Therefore I will merely suggest some other possible objects, a few of which have actually been implemented by myself and others.

The first is a horizontal line with an optional embedded title string. The trick is to make the object's horizontal size very large — perhaps 4000 pixels — so it always will occupy a line all by itself. The draw handler draws the line and then superimposes the title string.

The second is an icon which pops up an annotation balloon when you click and hold on it. The click handler simply pops up a help balloon containing a string taken from the object's data handle.

Glenn M. Berntson did an outline object. This is a little tricky… first, he had to find a way to expand or contract the outline at any level, and second, he had to indent everything to the proper depth. The main trick is that when an outline level is collapsed, all the contained text and lower levels must be excised from the text itself and stashed away in the uplevel object's data handle. The process is reversed for expanding. In WASTE 2.0 (which Glenn used) it's possible to use paragraph-level formatting to handle indentation; in WASTE 1.3 it would probably be necessary to install special line-formatting hooks.

An "eyes" object with different facial expressions would be a neat trick. Once you update the object continuously in your event loop, you can have the eyes always looking at the cursor. Refinements might have the object getting cross-eyed if the cursor passes over it or even, perhaps,

growing progressively more panic-stricken as the dreaded insertion point approaches, and finally running for it's life as doom closes in! Help!!

Shared objects. The idea here would be to have several instances of the same object, perhaps an icon or picture resource, and have the object's data handle just contain the resource ID. This is useful for manuals, where you use the same simple graphics over and over, and storing dozens of identical pictures would be wasteful.

Final Words

There, that wasn't so hard, was it? As you saw, it's possible find interesting, amusing and even valuable objects in what at first sight appears to be a WASTE.

You may use the enclosed source code in your application at no cost; however, I'd appreciate getting credit in your "About Box". Be warned, however, that the objects were either specifically written only for inclusion in this paper or are shortened versions from my own applications; in any event, none were submitted to industrial-strength testing — and indeed, some haven't even been compiled in the form that they appear! All such use is at your own risk. The best defense is to thorougly understand what's going on.

Special thanks go to the creator of WASTE, Marco Piovanelli, WASTE demigods Dan Crevier and Timothy Paustian, and to my fellow explorers from the WASTE mailing list.

I'd also like to thank all of my family, wherever and whoever they may be, and indeed everybody which I've ever met or heard of… you all have been valuable teachers.

Appendix A: Our Hero, WEClick, on Steroids

This replaces the original routine in "WEMouse.c".

```
pascal void WEClick(Point mouseLoc, EventModifiers modifiers, UInt32 clickTime,
WEHandle hWE)
{
      WEPtr pWE;
      LongPt thePoint;
      SInt32 offset, anchor;
      SInt32 rangeStart, rangeEnd;
      WEEdge edge;
      Boolean isMultipleClick;
      Boolean saveWELock;
#if WASTE_DRAG_AND_DROP
      Boolean notDrag=true;
#endif
#if WASTE_OBJECTS
      WEObjectDescHandle hObjectDesc;
#endif
#if WASTE_IC_SUPPORT
      SInt32 urlStart, urlEnd;
#endif

      // stop any ongoing inline input session
      WEStopInlineSession(hWE);
```

```
	// lock the WE record
	saveWELock = _WESetHandleLock((Handle) hWE, true);
	pWE = *hWE;

#if WASTE_IC_SUPPORT
	// remember the selection range before the click
	urlStart = pWE->selStart;
	urlEnd = pWE->selEnd;
#endif

	// hide the caret if it's showing
	if (BTST(pWE->flags, weFCaretVisible))
	{
		_WEBlinkCaret(hWE);
	}

	// find click offset
	WEPointToLongPoint(mouseLoc, &thePoint);
	offset = WEGetOffset(&thePoint, &edge, hWE);

	// determine whether this click is part of a sequence
	isMultipleClick = ((clickTime < pWE->clickTime + GetDblTime())
				&& (offset == pWE->clickLoc));

	// remember click time, click offset and edge value
	pWE->clickTime = clickTime;
	pWE->clickLoc = offset;
	pWE->clickEdge = edge;

	if ((modifiers & shiftKey) == 0)
	{

		// is this click part of a sequence or is it a single click?
		if (isMultipleClick)
		{
			pWE->clickCount++;

			// a double (triple) click creates an
			// anchor-word (anchor-line)
			if (pWE->clickCount > 1)
			{
				WEFindLine(offset, edge, &pWE->anchorStart,
					&pWE->anchorEnd, hWE);
			}
			else
			{
				WEFindWord(offset, edge, &pWE->anchorStart,
					&pWE->anchorEnd, hWE);
			}

			offset = pWE->anchorStart;
		}
		else
		{
			// single-click

#if WASTE_DRAG_AND_DROP
			// if the Drag Manager is available and the click
```

```
                  //    went in the selection range,
                  // this click may be the beginning of a drag gesture
                  if (BTST(pWE->flags, weFHasDragManager)
                        && BTST(pWE->features, weFDragAndDrop))
                  {
                        if (_WEOffsetInRange(offset, edge, pWE->selStart,
                              pWE->selEnd))
                        {
                              notDrag = false;
                              if (_WEDrag(mouseLoc, modifiers, clickTime,
                                    hWE) != weNoDragErr)
                              {
                                    goto cleanup;
                              }
                        }
                  }
#endif

                  pWE->clickCount = 0;
                  anchor = offset;
            }
      }
      else
      {

            // if the shift key was down, use the old anchor offset
            //    found with the previous click
            anchor = BTST(pWE->flags, weFAnchorIsEnd) ?
                        pWE->selEnd : pWE->selStart;
      }

      // set the weFMouseTracking bit while we track the mouse
      BSET(pWE->flags, weFMouseTracking);

      // MOUSE TRACKING LOOP
      do
      {

            // get text offset corresponding to mouse position
            WEPointToLongPoint(mouseLoc, &thePoint);
            offset = WEGetOffset(&thePoint, &edge, hWE);

            // if we're selecting words or lines, pin offset to a
            //    word or line boundary
            if (pWE->clickCount > 0)
            {
                  if (pWE->clickCount > 1)
                  {
                        WEFindLine(offset, edge, &rangeStart, &rangeEnd, hWE);
                  }
                  else
                  {
                        WEFindWord(offset, edge, &rangeStart, &rangeEnd, hWE);
                  }

                  // choose the word/line boundary and the anchor
                  //    that are farthest away from each other
                  if (offset > pWE->anchorStart)
```

```
                {
                        anchor = pWE->anchorStart;
                        offset = rangeEnd;
                }
                else
                {
                        offset = rangeStart;
                        anchor = pWE->anchorEnd;
                }
        }
        else
        {
                // if the point is in the middle of an object, the
                //    selection should include it
                if (edge == kObjectEdge)
                {
                        offset++;
                }
        }

#if WASTE_OBJECTS
        if (((modifiers & shiftKey) == 0)
#if WASTE_DRAG_AND_DROP
                && notDrag
#endif
                && !isMultipleClick)
        {
                rangeStart = anchor - (edge == kTrailingEdge);
                rangeEnd = offset + (edge == kLeadingEdge);
                if ((rangeStart >= 0) && (rangeEnd <= pWE->textLength)
                        && (rangeEnd - rangeStart == 1))
                {
                        Rect frame;
                        WERunInfo runInfo;
                        WEGetRunInfo(rangeStart, &runInfo, hWE);
                        if ((hObjectDesc = runInfo.runAttrs.runStyle.tsObject)
                                != nil)
                        {
                                WEGetObjectFrame(hObjectDesc, &frame);
                                GetMouse(&mouseLoc);
                                if (PtInRect(mouseLoc, &frame) &&
                                        _WEClickObject(mouseLoc,
                                        modifiers | ((anchor + 1) == offset ? 2 : 0),
                                        clickTime, hObjectDesc))
                                {
                                        pWE->clickLoc = kInvalidOffset;
                                        break;
                                }
                        }
                }
        }
#endif

        // set the selection range from anchor point to current offset
        WESetSelection(anchor, offset, hWE);

        // call the click loop callback, if any
        if (pWE->clickLoop != nil)
```

```
            {
                  if (!CallWEClickLoopProc(hWE, pWE->clickLoop))
                  {
                        break;
                  }
            }

            // update mouse position
            GetMouse(&mouseLoc);

      } while(WaitMouseUp());

      // clear the weFMouseTracking bit
      BCLR(pWE->flags, weFMouseTracking);

      // redraw the caret immediately if the selection range is empty
      if (anchor == offset)
      {
            _WEBlinkCaret(hWE);
      }

#if WASTE_IC_SUPPORT
      if (modifiers & cmdKey)
      {
            // command+clicking a URL tries to resolve it
            // we normally ask IC to parse the text
            //    surrounding the clicked point,
            //    but if a selection already existed prior to the click, we
            /     /pass that to IC rather than forcing a re-parse
            if ((anchor != offset) || (anchor < urlStart) ||
                  (anchor > urlEnd))
            {
                  urlStart = anchor;
                  urlEnd = offset;
            }
            _WEResolveURL(modifiers, urlStart, urlEnd, hWE);
      }
#endif

cleanup:
      // unlock the WE record
      _WESetHandleLock((Handle) hWE, saveWELock);

}
```

Appendix B: WESetObjectSize and its Faithful Sidekick, WEGetObjectOffset

The first routine is actually a hybrid between my own original code and the same routines from WASTE 2.0a3. The second is entirely from 2.0; Marco's solution was much better than my own. Paste them into "WEObjects.c", and remember to make the necessary modifications to both header files: "WASTE.h" and "WASTEIntf.h".

```
pascal OSErr WESetObjectSize(WEObjectDescHandle hObjectDesc, Point objectSize)
{
      WEHandle hWE = (*hObjectDesc)->objectOwner;
      WEPtr pWE;
```

```
SInt32 offset;
Boolean saveWELock;
OSErr err;

// lock the WE record
saveWELock = _WESetHandleLock((Handle) hWE, true);
pWE = *hWE;

// return an error code if this instance is read-only
err = weReadOnlyErr;
if (BTST(pWE->features, weFReadOnly))
{
      goto cleanup;
}

// find the object's offset within the text
offset = WEGetObjectOffset(hObjectDesc);

// return an error code if the object can't be found
err = weObjectNotFoundErr;
if (offset == kInvalidOffset)
{
      goto cleanup;
}

//    do nothing if the size didn't change
err = noErr;
if ((*hObjectDesc)->objectSize == objectSize)
{
      goto cleanup;
}

// stop any ongoing inline input session
WEStopInlineSession(hWE);

// increment modification count
pWE->modCount++;

// if undo support is enabled, save object range
if (BTST(pWE->features, weFUndoSupport))
{
      WEClearUndo(hWE);
      if (_WENewAction(offset, offset + 1, 0, weAKUnspecified, 0,
            hWE, &hAction) == noErr)
      {
            _WEPushAction(hAction);
      }
}

// change the object size
(*hObjectDesc)->objectSize = objectSize;

// redraw the affected range
if ((err = _WERedraw(offset, offset + 1, hWE)) != noErr)
{
      goto cleanup;
}
```

```
	//	clear result code
	err = noErr;

cleanup:
	// unlock the WE record
	_WESetHandleLock((Handle) hWE, saveWELock);

	// return result code
	return err;
}

pascal SInt32 WEGetObjectOffset(WEObjectDescHandle hObjectDesc)
{
	WEPtr pWE = *(*hObjectDesc)->objectOwner;
	WEStyleTableEntry * pStyle = *pWE->hStyles;
	WERunArrayEntry * pRun = *pWE->hRuns;
	SInt32 nStyles = pWE->nStyles;
	SInt32 nRuns = pWE->nRuns;
	SInt32 styleIndex;
	SInt32 runIndex;

	//	look for this object in its owner's style table
	for ( styleIndex = 0 ; styleIndex < nStyles ; styleIndex ++ )
	{
		if ( pStyle->info.runStyle.tsObject == hObjectDesc)
		{
			break;
		}
		pStyle ++ ;
	}

	//	look for the style table index in the style run array
	for ( runIndex = 0 ; runIndex < nRuns ; runIndex ++ )
	{
		if ( pRun->styleIndex == styleIndex )
		{
			return pRun->runStart;
		}
		pRun ++ ;
	}

	//	object not found: this should never happen
	return kInvalidOffset;
}
```

Appendix C: Who Framed Roger Object?

Paste this routine into "WEObjects.c", and remember to make the necessary modifications to both header files: "WASTE.h" and "WASTEIntf.h".

```
pascal void WEGetObjectFrame(WEObjectDescHandle hObjectDesc, Rect* frame)
{
	WEHandle hWE = WEGetObjectOwner(hObjectDesc);
	SInt32 offset = WEGetObjectOffset(hObjectDesc);
	Point size = WEGetObjectSize(hObjectDesc);
	Point where;
```

```
	LongPt whereLong;
	WELineRec *pLine;
	SInt32 lineIndex;
	if (offset == kInvalidOffset)
	{
		frame->top = frame->left = 0;
		frame->right = size.h;
		frame->bottom = size.v;
	}
	else
	{
		lineIndex = WEOffsetToLine(offset, hWE);
		pLine = *(*hWE)->hLines + lineIndex;
		WEGetPoint(offset, WEGetDirection(hWE), &whereLong, nil, hWE);
		WELongPointToPoint(&whereLong, &where);
		frame->bottom = where.v + pLine->lineAscent;
		frame->top = frame->bottom - size.v;
		frame->left = where.h;
		frame->right = where.h + size.h;
	}
}
```

Appendix D: Links for Further Explorations

http://www.merzwaren.com/waste/ is Marco Piovanelli's official WASTE page. Releases, documentation, license terms, and so forth. Check regulary for news about the WASTE 2.0 release.

http://www.boingo.com/waste/ is Dan Crevier's compreensive index of WASTE links, WASTE-using applications, and related subjects. Dan also maintains the WASTE mailing list. Check this page for subscription details.

http://www.bact.wisc.edu/CWASTEEdit/CWASTEEdit.html is about Timothy Paustian's wrapper class for use with PowerPlant. It will be superseded soon after WASTE 2.0 comes out by Tim's new WText class.

You can e-mail me at mailto:rainer@ez-bh.com.br, or through my web page at http://www.ez-bh.com.br/~rainer/