Adobe Technical Journal

Making a plug-in scripting-aware for Photoshop 4.0 Rev. 2

Andrew Coven
Photoshop Developer Support Engineer
Adobe Systems, Incorporated

gapdevsup@adobe.com

1.0 Abstract
Making a plug-in scripting-aware for Adobe Photoshop 4.0

The Adobe® Photoshop® 4.0 application programming interface introduces a new feature for automation: *actions*. Controlled by the user via the *actions palette*, plug-ins can execute pre-defined commands and batches to allow the user to automate routine and difficult tasks from a single button-click. This article details the process used to update two Adobe Photoshop 3.0.5 plug-ins, *Dissolve* and *DummyScan* (which was renamed *GradientImport*), to make them scripting-aware and controllable via the actions palette.

2.0 Introduction
Welcome to Adobe Photoshop 4.0 Actions

The Adobe® Photoshop® 4.0 application programming interface (API) extends the 3.0.5 specification to include a number of new items. One that affects all the plug-in types and specifications is the new automation system. The main user interface for the automation system is the *actions palette*. The actions palette allows the user to specify commands and plug-ins that are scripting-aware and record multiple events into actions that can be executed with a single mouse-click.

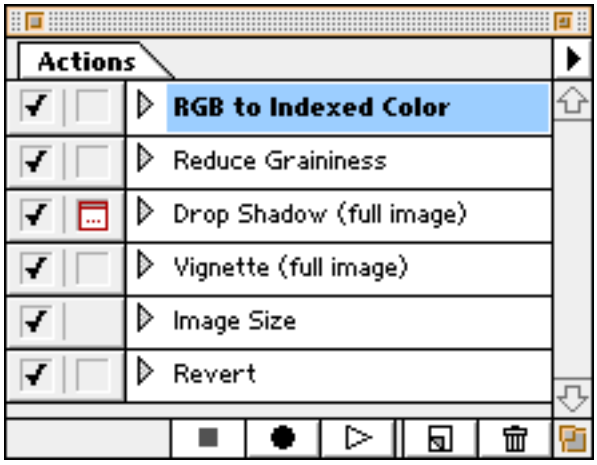

Figure 1: Actions palette

A folder or group of files can also be controlled so that actions can be applied in a batch. This is called *batch-processing* and is part of the Adobe Photoshop 4.0 actions palette.

All plug-ins can be controlled by the scripting system as execute-only commands. This means, whether the plug-in is *scripting-aware* or not, the action system can execute the plug-in as if the user had invoked it from its menu.

A scripting-aware plug-in, however, goes further, and allows the action system to control your plug-in's parameters automatically. This means that, unless there is an error or a parameter that your plug-in needs that it didn't get, your plug-in can operate silently, not needing to show its user interface and interact with the user. This is extremely valuable for batch-processing and generating special effects that require numerous commands and parameters.

2.1 Converting 3.0.5 to 4.0

My task was to take the existing plug-ins that shipped with the 3.0.5 software development kit (SDK) and convert them all to the 4.0 API spec. This proved to be fairly straight-forward for some plug-in types, such as simple filters, and more involved for others, such as Import modules, especially with ones that do multiple imports.

This article will detail how I converted two plug-ins, the Filter plug-in module *Dissolve* and the Import plug-in module, *GradientImport*, to be scripting-aware.

The filter plug-in was vastly simpler, so I'll start with that, and then detail the process for GradientImport, which required additional code to handle the multiple import routines.

2.2 Scope of this article

2.2.1 More detail is in the SDK

Intimate details on all the scripting parameters and callback suites are available in the Adobe® Photoshop® 4.0 SDK, which is available at Adobe's web site:

```
http://www.adobe.com/supportservice/devrelations/sdks.html
```

This article will only address the callbacks and structures that were pertinent to updating the two plug-in example modules. There is much more to the scripting system than is covered in this document. I recommend you read the SDK for more detail.

2.2.2 Macintosh or Windows?

Scripting implementation, recording, and playback are all part of the Adobe Photoshop API. This means that, except in a few rare exceptions, the callbacks, data structures, and parameters are all exactly the same on both Macintosh and Windows. This article shows Macintosh user interface examples, but the discussion and examples are comparable, if not exactly the same, on Windows.

3.0 Starting out

3.1 Basic scripting approach

The approach to creating a scripting-aware plug-in is detailed in the scripting chapter of the Photoshop SDK programmer's guide:

1. Look at your user interfaces and describe the parameters as human-readable text;

2. Create a terminology resource for your plug-in and your PiPL HasTerminology property;

3. Update your plug-in code to record scripting events and objects;

4. Update your plug-in code to be automated by (playback) scripting events and objects.

With this in mind, I looked at the user interface for the Dissolve filter. This was the same both on Macintosh and Windows. The Macintosh interface is shown in Figure 2.

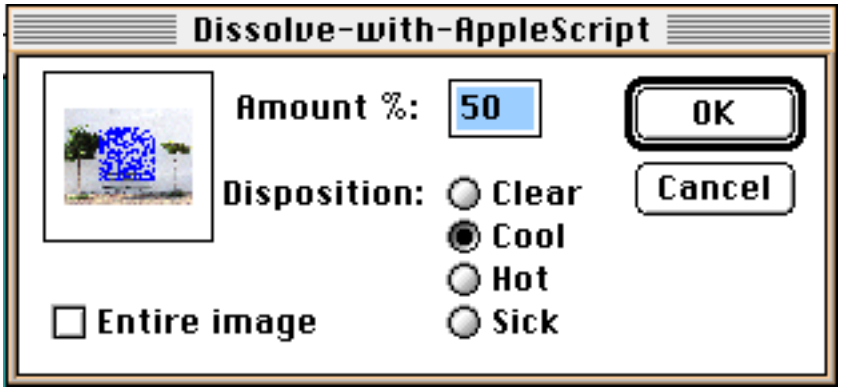

Figure 2: Dissolve filter user interface

After looking at my interface, I was able to describe it as these elements:

1. A button, “OK”, which I don’t need to be recordable.

2. A button, “Cancel”, which I don’t need to be recordable.

3. An amount, expressed as an integer from 1 to 100 representing a percentage.

4. A disposition, expressed as a textual enumeration of a mutually-exclusive list of options, either “Clear”, “Cool”, “Hot”, or “Sick”.

5. A flag for “entire image”, expressed as a boolean value of either yes or no.

This should look familiar. It is reminiscent of the resource text used to describe Macintosh dialog items.

When describing these items, it’s important to keep in mind how they will look when represented in the actions palette. Since the actions palette does get loaded with text, it makes sense to use single labels whenever possible and where it will be more readable to the user. I could have used four booleans for “Clear”, “Cool”, “Hot”, and “Sick”, but since “Disposition” should always only be one thing, it makes more sense to have the actions palette display:

Dissolve
Amount: 20%
Disposition: Cool

Than something like:

Dissolve
Amount: 20%
without Clear
with Cool
without Hot
without Sick

And speaking of booleans, it's usually much better form to always leave the default value of a boolean as implied instead of explicitly showing it in the actions palette. Again, because the palette can get pretty large, it's better to only store boolean values that are different than your default. For instance, in the example above, "Entire Image" isn't listed in the palette because it was in its default (off) state. If it is checked, then I would store it in the action descriptor and it would get displayed as:

Dissolve
Amount: 20%
Disposition: Cool
with Entire Image

4.0 Creating a terminology resource

4.1 AppleScript/AppleEvents

AppleScript and AppleEvents are the Macintosh's automation system. The Photoshop 4.0 scripting system is based heavily on the programming architecture defined by Apple. Most users think of AppleScript and AppleEvents from the user perspective: the Macintosh script editor, firing off events to different applications to automate procedures. What I'll be describing here is the internal workings necessary to define events to an external system. In this case, the plug-ins, such as Dissolve, must take on extra descriptors that make their parameter's available to the host, in this case, Adobe Photoshop 4.0. The terminology resource is the first internal description system that bridges the gap between the plug-ins programming parameters and the external automation system.

And, as stated, the Photoshop 4.0 automation system, while designed around the AppleScript/AppleEvent model, has been created to integrate fully with OLE Automation on Windows. More information on that is available in the appendix of the Photoshop SDK Guide.

4.2 Start with the examples

The terminology resource is a standard AppleScript/AppleEvent '`aete`' resource. The terminology resource is a bit cumbersome, so I always

recommend starting with the example code. In this case, I had to make it from scratch. First, I chose to define some common parameters that would change from plug-in to plug-in:

```
#define vendorName    "AdobeSDK"          // unique vendor name
#define ourSuiteID    'sdK1'              // must follow id guidelines
#define ourClassID    ourSuiteID          // must be unique, but can be suite id
#define ourEventID    'disS'              // must follow id guidelines
#define ResourceID    16000               // typical id for plug-ins
#define uniqueString  ""                  // empty
```

Then, I created the terminology resource:

```
resource 'aete' (ResourceID, purgeable)
{  // aete version and language specifiers:
   1, 0, english, roman,
   {  // vendor suite name:
      vendorName,                         // "AdobeSDK"'
      "Adobe example plug-ins",           // optional description
      ourSuiteID,                         // suite id 'sdK1'
      /* This is extremely important. All IDs, keys, and names must be unique. The SDK
      describes a naming convention that must be followed explicitly. Your scripting keys
      and IDs (unsigned32) must always follow these rules:
      1. They must start with a lowercase letter.
      2. They must contain at least one uppercase letter.
      3. They cannot be all lowercase.
      4. They cannot be all uppercase.
      More below when we get to keys. */
      1,                                  // suite code, must be 1
      1,                                  // suite level, must be 1
      {  // structure for filters. Unique filter name:
         vendorName " dissolve",          // "AdobeSDK Dissolve"
         "dissolve noise filter",         // optional description
         ourClassID,
         // class id must be unique or suite id. Suite id 'sdK1'.
         ourEventID,                      // unique event id 'disS'

         NO_REPLY,                        // never a reply
         IMAGE_DIRECT_PARAMETER,
         // direct parameter. See PIActions.h for other macros.
         {  // parameters:
            "amount",                     // parameter name
            /* must be predefined parameter name and key from PIActions.h or unique
            name and key id. See 'disposition' for example. */
            keyAmount,                    // parameter key
            /* must be predefined parameter key from PIActions.h or unique key id. */
            typeFloat,                    // parameter type
            // typeInteger, typeBoolean, typeText, etc., all defined in PIActions.h
```

```
            "dissolve amount",              // optional description
            flagsSingleParameter,           // parameter flags
            // Other parameters in PIActions.h

            // Second parameter:
            vendorName " disposition",
            // unique name "AdobeSDK disposition"
            keyDisposition,                 // unique key 'disP'
            typeMood,                       // unique type 'mooD'
            "dissolve disposition",         // optional description
            flagsEnumeratedParameter        // parameter flags for enum

            vendorName " entire image",
            // unique name "AdobeSDK entire image"
            keyEntireImage,                 // unique key 'entI'
            typeBoolean,
            flagsSingleParameter
         } // close parameters
      }, // close filter structure
      {}, // plug-in classes for all other plug-ins here (we'll use this later)
      {}, // comparison ops (not supported)
      { // any enumerations. We have one, typeMood:
         typeMood,                          // unique type 'mooD'
         {
            vendorName " clear",
            // unique name "AdobeSDK clear"
            dispositionClear,               // unique key 'moD0'
            "clear headed",                 // optional description

            vendorName " cool",
            // unique name "AdobeSDK cool"
            dispositionCool,                // unique key 'moD1'
            "got the blues",                // optional description

            vendorName " hot",
            // unique name "AdobeSDK hot"
            dispositionHot,                 // unique key 'moD2'
            "red-faced",                    // optional description

            vendorName " sick",
            // unique name "AdobeSDK sick"
            dispositionSick,                // unique key 'moD3'
            "green with envy"               // optional description
         } // close typeMood
      } // close enumerations
   } // close vendor suite
}; // close 'aete'
```

The terminology resource is parsed on the Macintosh side by a standard template included with most compilers. On the Windows side, it is precompiled along with the 'PIPL' resource and then parsed by the Photoshop resource file converter, CNVTPIPL.EXE. Either way, the Dissolve.r file is converted into a working resource that is used at runtime by the host.

4.3 Add the HasTerminology resource to your PiPL

Once I had a complete terminology resource, I have to tell Photoshop where to find it, since a single plug-in file can have multiple modules in it. To do that, a new PiPL type has been added, *HasTerminology*. It's syntax is:

```
HasTerminology { ourClassID, ourEventID, ResourceID, uniqueString }
```

Just to review, in the case of examples, I defined:

```
#define vendorName   "AdobeSDK"        // unique vendor name
#define ourSuiteID   'sdK1'            // must follow id guidelines
#define ourClassID   ourSuiteID        // must be unique, but can be suite id
#define ourEventID   'disS'            // must follow id guidelines
#define ResourceID   16000             // typical id for plug-ins
#define uniqueString ""                // empty
```

The AppleScript and AppleEvent architecture makes all key and name dictionaries global, which is why unique key/name pairs are required. A predefined dictionary of common terms is defined in PIActions.h. You can use those keys and their obvious names (keyColor, name "Color") instead of having to create unique key and name pairs. I recommend using the standard keys whenever you possibly can.

If you define a uniqueString, then your plug-in will stay scoped only to Photoshop and will not have to worry about having globally unique names. But you still have to worry about conflicting with your own other suites using that same uniqueString. This means that I would not have had to use key names such as "AdobeSDK disposition"—I could have just used "disposition." I chose to keep everything scoped globally for future AppleScript/AppleEvent compatibility.

5.0 Creating a scripting recording function

The next step for Dissolve was to record my parameters. There are a number of utility routines defined in PIUtilities.h and PIUtilities.c to make reading and writing from descriptors easier than having to access the procedures directly through the callback structure. You cannot check a scripting playback function, nor whether a terminology resource is correct, until some parameters are handed to Photoshop.

5.0.1 To use globals or not to use globals, that is the question!

For versions of Photoshop prior to 4.0, the only way to track global variables was for you to allocate the memory yourself and store the global values in a parameter handle that was handed back to the plug-in on subsequent interations.

The Photoshop 4.0 scripting system will always pass your plug-in a *descriptor* at every selector call. A descriptor is a set of keys and values, very much like a set of predefined global values. Theoretically, I could use the scripting system to track my global values, instead of passing my entire global struct to my different routines and storing it in the parameter handle.

To make that change, I'd have to take out all my global variables and change to reading and storing my parameters in the scripting descriptor on every selector call. That's a lot of work, and I didn't feel I gained anything from that.

Instead, I decided to stay with my global routines, and use the scripting system to write out my final values and read in values to override my initial global values. This made much more sense, and allows the plug-ins to operate in a non-scripting environment, such as older versions of Photoshop.

5.1 WriteScriptParams routine

I created a routine, `WriteScriptParams`, that took the global values and created a descriptor to hand back to the host.

I created a new source file, `DissolveWithScripting.c`, to hold the playback and recording script functions.

```
OSErr WriteScriptParams (GPtr globals)
{
   double               percent = gPercent;
   /* I'm using a double because I want to use scripting type UnitFloat with unitPercent,
   which is a double value. By using UnitFloat, my value will display in the actions palette
   with a percent sign after it. Cool! */
   PIWriteDescriptor    token = nil;
   OSErr                gotErr = noErr;

   if (DescriptorAvailable())
   {
      /* DescriptorAvailable() is a macro from PIUtilities that checks to see if the
      gStuff–>descriptorParameters callback parameter block is available. */

      token = OpenWriter();
      // OpenWriter() is a macro from PIUtilities that creates a new write descriptor.

      if (token)
```

```
        {  // we got a valid token to work with. Write our keys:
            PIPutUnitFloat(token, keyAmount, unitPercent, &percent);
            /* this is a macro from PIUtilities. It requires the token to write to, the key, the
            unit (unitPercent, unitDistance, unitPixels, etc., defined in PIActions.h), and
            then a pointer to the double. */

            PIPutEnum(token, keyDisposition, typeMood, gDisposition);
            /* another macro from PIUtilities. This writes an enumeration. It takes the token,
            the key, the list of enumerations (the type) and the actual enumeration.
            gDisposition is an unsigned32 that is either dispositionClear, dispositionCool,
            dispositionHot, or dispositionSick. Note that if these weren't defined in the
            terminology resource, it would display nothing, or garbage. The enum stored
            must match the keys in the enumeration list in the 'aete'.*/

            if (gIgnoreSelection)
                PIPutBool(token, keyEntireImage, gIgnoreSelection);
                /* Like I suggested, when you are writing boolean values, it makes the actions
                palette look cleaner if you only write them when they are in their non-default
                value. In this case, when gIgnoreSelection is true (the default is to use the
                selection) then the macro from PIUtilities writes the key and boolean value to
                the descriptor in token. */

            gotErr = CloseWriter(&token);
            /* This is a very useful routine defined in PIUtilities. When you close a token, it
            returns with a handle to a descriptor. This descriptor is then what you pass to the
            host for it to display in the actions palette (and subsequently return to you on
            playback.) CloseWriter closes the token and stores the descriptor in the
            gStuff–>descriptorParameters callback parameter block, which is how a plug-in
            hands back a descriptor. It then deallocates token and sets it to null. Lastly, it sets
            the recordInfo parameter to dialogOptional, which is the standard return value to
            tell the host "Only pop my dialog when the user wants it." For a description of
            recordInfo, see the Scripting chapter of the SDK and PIUtilities.*/
        } // close token
    } // close DescriptorAvailable
    return gotErr;
} // end WriteScriptParams
```

5.2 Calling WriteScriptParams

I call `WriteScriptParams` in `DoFinish`, as that's the last routine the plug-in executes before it completely returns to the host.

5.3 Running the plug-in and errors in scripting

Once I completed my `WriteScriptParams` routine, it was time to try it out to see if the terminology resource, HasTerminology PiPL property, and `WriteScriptParams` routine worked. I did this by placing an alias to the

plug-in in the Photoshop plug-ins directory, deleting my preferences file (to start fresh) and running Photoshop.

I then opened a document and clicked the “document” icon in the actions palette, which is the “New Action” button. I named it, then went to my plug-in and executed it with some basic parameters. Finally, I clicked the “stop” button in the actions palette, and checked to see if my plug-in had been recorded.

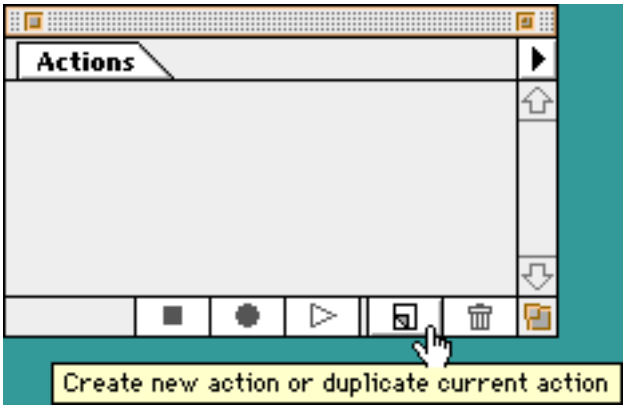

Figure 3: Creating a new action in the actions palette

Here is a list of issues and answers I found in debugging from this step:

5.3.1 My plug-in wasn’t in the filters menu.

This happened when I didn’t put the plug-in in the right directory, that Adobe Photoshop was loading plug-ins from the preferences file (and not scanning the directory to look for new plug-ins), or that my PiPL resource wasn’t valid.

5.3.2 My plug-in didn’t get recorded.

This was usually because I wasn’t handing back a proper descriptor. I was either handing back null, accidentally, or I was storing garbage data in the descriptor which was messing everything up.

5.3.3 The actions palette says my plug-in’s name, but none of its parameters (such as “Using: Dissolve” but nothing else)

This means scripting system did not find a valid '`aete`' dictionary resource, and/or it did not find a valid reference to the resource in the HasTerminology property. It’s usually either a bad reference number in the HasTerminology property, a bad construction of the HasTerminology property, or a badly

formed dictionary resource. On the Macintosh side, the resource compiler will complain if the dictionary resource of `Dissolve.r` is not formed properly. On the Windows side, `CNVTPIPL.EXE` will complain. Unfortunately, neither will complain if the keys and data you hand back in your descriptor do not match the keys in your dictionary resource. It just won't display.

5.3.4 The actions palette displays labels with no data after them, such as "Amount: %"

This was due to a messed up descriptor. I was either handing back invalid (or improper) data (such as mixing up my keys and data types) or I was handing back no descriptor (accidentally handing back null, for instance.)

5.3.5 The actions palette displays labels with scrambled data

This happened when I had different keys in my dictionary than I was storing in my descriptor. If I had a `typeInteger` for `keyAmount` but then stored using `typeFloat`, or if I was storing `typeText` and passed binary instead of alphanumeric information in the handle.

5.4 Actions palette with Dissolve action

Figure 4 shows the actions palette once I got the proper descriptor recorded, along with good dictionary and HasTerminology resources.

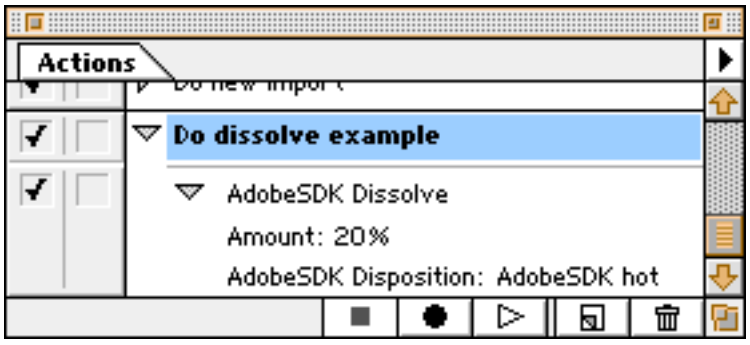

Figure 4: Dissolve filter actions palette display

6.0 Automating the plug-in for playback

Now that the plug-in was correctly recording and displaying a descriptor, it was time to prepare it to read that descriptor when it was handed to me, and honor those parameters.

Taking the same approach to globals as the `WriteScriptParams` routine, I created a `ReadScriptParams` routine, with the purpose of opening, pulling keys and values out of a descriptor, and overriding the global variables.

```
Boolean ReadScriptParams (GPtr globals)
{
   double               x = 0;
   const double         minValue = kPercentMin, maxValue = kPercentMax;
   // used to pass minimum and maximum values for PinUnitFloat
   unsigned long        percentUnitPass = unitPercent;
   // used to pass unitPercent to PinUnitFloat

   PIReadDescriptor     token = NULL;
   DescriptorKeyID      key = NULLID;
   DescriptorTypeID     type = NULLID;
   int32                flags = 0;
   DescriptorKeyIDArray array = { keyAmount, keyDisposition, NULLID };
   /* this array will be checked off as each key is read. It should return { keyNull, keyNull,
   NULL }. If it doesn't, then we've missed a key somewhere. See errMissingParameter,
   below. */

   OSErr                stickyError = noErr;
   Boolean              returnValue = true;
   // ReadScriptParams returns with whether to pop the dialog or not (true = show dialog)

   if (DescriptorAvailable())
   { // If descriptorParameters callback suite is available, do this:
      token = OpenReader(array);
      /* routine from PIUtilities. Opens the descriptor pointed to in
      gStuff–>descriptorParameters, starts tracking keys in array, and returns a read token
      to work with. */
      if (token)
      { // got a valid read token. Now start grabbing keys until we get null:
         while (PIGetKey(token, &key, &type, &flags))
         { // we got a valid (non-null) key. See which value it is:
            switch (key)
            { // we can receive these keys in any order, so check to see which one:
               case keyAmount:
                  PIGetPinUnitFloat(token, &minValue, &maxValue,
                  &percentUnitPass, &x);
                  /* this is a routine from PIUtilities. It gets a unit-delimited value
                  (such as unitPixels, unitPercent) and automatically pins it between
                  minValue and maxValue. The value is returned in the last parameter,
                  which is the address of a double (in this case, "x"). If the value had to
                  be coerced (pinned to the low or high number) then this routine will
                  return the coercedParam error, but "x" will still be a valid number. */
                  gPercent = x; // assign to our global
                  break;
```

```
                case keyDisposition:
                    PIGetEnum(token, &gDisposition);
                    /* this is another routine from PIUtilities. It reads an enumerated
                    value. Since our global is an unsigned32, we can have PIGetEnum
                    store the value directly to the global. */
                    break;

                case keyEntireImage:
                    PIGetBool(token, &gIgnoreSelection);
                    /* from PIUtilities, returns a boolean value. Since our global is a
                    boolean, we pass its address and have it set directly. */
                    break;

                // ignore all other cases and classes
            }
        }

        stickyError = CloseReader(&token);
        /* CloseReader, from PIUtilities, automatically closes the read token, deallocates
        it, and stores null in token. It returns an error code, indicating if any errors were
        encountered during the getKey routine.

        if (stickyError)
        {
            if (stickyError == errMissingParameter)
                ; /* errMissingParameter = -1715, which means one of the keys in
                descriptorKeyIDArray was not found. Walk the array, and whatever is not
                "typeNull" is the value not found in the descriptor. For this example, I
                can go with the default values if I missed a key. If you cannot, or cannot
                coerce a value from the keys you did receive, then you might want to show
                your dialog. Whether or not you can show your dialog depends on
                PlayDialog(). See below. */
            else
                gResult = stickyError; // we got a real error. Report it.
            } // close stickyError
        } // close token
        gQueryForParameters = returnValue = PlayDialog();
        /* PlayDialog() examples playInfo inside gStuff–>descriptorParameters and returns
        true if it is plugInDialogDisplay, which means "please display your dialog." If it is
        plugInDialogSilent, you must never show your dialog, and if it is
        plugInDialogDontDisplay, then don't display your dialog unless you need to. (Such as
        if you missed a key you need and cannot coerce.) */
    } // close descriptorAvailable
    return returnValue; /* the global variable gQueryForParameters determines
    whether I need to pop my dialog, but I'll return this value, as well. */
} // end ReadScriptParams
```

6.1 Calling ReadScriptParams and ValidateParameters

Calling `ReadScriptParams` is a little trickier. I want to call it after I've initialized my globals, but before I need them. Sometimes, however, my plug-in may be called and I may never get to the `DoParameters` routine, which initializes my globals. This happens in Adobe Premiere, which only executes the plug-in completely once, then passes its parameters in for every frame of a filmstrip. This also can occur when a plug-in has been recorded, then the user quits Photoshop, runs it again, and executes the action right from the palette. Literally, I may go to store values in my globals before I've allocated space for them. Because of this danger, I decided to pull some of the initialization routines out of `DoParameters` and create an additional routine, `ValidateParameters`, which checks to see if the parameters are valid, and if not, initializes them. That way I can call it right at the beginning of my `DoStart` routine, right before I dispatch to my user interface and code which depends on my globals.

Anywhere before `DoStart` that I might use my globals, I need to check them for validity first. That could be in `DoParameters`, `DoPrepare`, or `DoStart`:

```
void DoParameters (GPtr globals)
{  /* Called on selectorParameters. We may not always get here on our first iteration (for
instance, if a user created an action calling this plug-in, quit Photoshop, then ran Photoshop
again and immediately executed the action. */

   ValidateParameters (globals);  // Check for valid parameters

   gQueryForParameters = TRUE;
   // If we're here, that means we're being called for the first time.
}
```

Now `ValidateParameters` does most of the work of `DoParameters`. This allows me to call it from multiple routines, to make sure my globals are valid and at least have default values before I use them:

```
void ValidateParameters (GPtr globals)
{ // Called whenever parameters need to be validated before used:
   if (!gStuff->parameters)
   { // Oops. Parameters haven't been allocated yet. Do that now.
      gStuff->parameters = NewHandle ((long) sizeof (TParameters));

      if (!gStuff->parameters)
      {  // Couldn't do it. Must be out of memory.
         gResult = memFullErr;
         return;
      }
      // Assign default global values:
      gPercent = 50;
      gDisposition = dispositionCool;
```

```
      gIgnoreSelection = false;
      gUseAdvance = false;
      gRowSkip = 1;
   } // close gStuff–>parameters
}
```

My DoPrepare routine does access some global variables, so I had to include a call to ValidateParameters before I used gRowSkip:

```
void DoPrepare (GPtr globals)
{ // Called on selectorPrepare to allocate memory requirements
   short                rowWidth = 0;
   short                total = 0;
   long                 oneRow = 0;
   long                 inOutRow = 0;
   long                 inOutAndMask = 0;

   gStuff->bufferSpace = 0;

   // Check maxSpace to determine if we can process more than a row at a time

   ValidateParameters (globals);
   // check to make sure gRowSkip has been initialized BEFORE we use it!

   total = gStuff->filterRect.bottom - gStuff->filterRect.top;
   rowWidth = gStuff->filterRect.right - gStuff->filterRect.left;

   oneRow = rowWidth * (gStuff->planes);
   // one row of data and its planes

   inOutRow = oneRow * 2; // inData, outData
   inOutAndMask = inOutRow + rowWidth;
   // maskData is only one plane (alpha)

   while (((inOutAndMask * gRowSkip) < gStuff->maxSpace) &&
         (gRowSkip < total))
      gRowSkip++;

   gStuff->maxSpace = gRowSkip * inOutAndMask; // all we need
}
```

Finally, right at the top of DoStart, I make a call to ValidateParameters to make sure, before I use my globals, that they've been at least assigned default values. Then I call ReadScriptParams to read the keys from the descriptor, if there is one, and override the default global values with the script parameters.

```
void DoStart (GPtr globals)
{ // Called from selectorStart. Main routine.
   ValidateParameters (globals);
   /* if stuff hasn't been initialized that we need, do it, then go check if we've got scripting
```

```
commands to override our settings */

ReadScriptParams (globals);
// update our parameters with the scripting parameters, if available

if (gQueryForParameters)
{ /* We got either plugInDialogDisplay or this is the first time the user has selected the
plug-in (so I have to pop the dialog to get the initial values) */
   PromptUserForInput (globals);        // Show the UI
   gQueryForParameters = FALSE;
}

// Rest of DoStart here.
```

6.2 Playback and recording questions: How do I know when...?

The obvious questions I had were:

"How do I know when I'm being played back?"
"When I'm being recorded?"
"When the user has selected me from the menu?"
"When the user has selected me in the actions palette?"

The answer to all of these is "You don't."

A plug-in has no way of knowing whether it's being recorded, played back, or directly interacted with by the user. This decision was made in the scripting implementation to make it as seamless with the original interface as possible. As long as you honor the `playInfo` flag, you will always know whether to pop your dialog or not. This includes if the user has clicked the **Dialog On** icon in the actions palette and is playing back your plug-in, or the user has selected your plug-in directly from the menu.

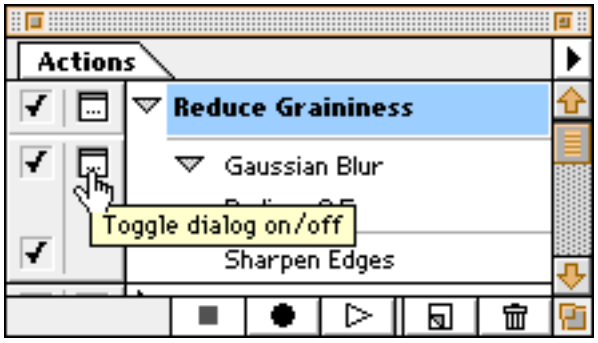

Figure 5: Toggle dialog option in actions palette

Whether the dialog has been requested or not, it makes sense to override any globals with any scripting keys provided before deciding to display the dialog

-- that way, the user can double-click to re-record an action and your plug-in will pop its dialog with the scripting parameters handed to it. Don't make the mistake (like I did, originally) of ignoring the scripting parameters just because `plugInDialogDisplay` has been requested. If it has been requested from within an action, like Figure 5, the user will expect to see the parameters from the actions palette in the plug-in's dialog.

Now that we're deep in the pool of scripting and you've gone through the simple example of the Dissolve filter plug-in, lets step up the complexity and look at an Import Module. In my case, it was the *DummyScan* example from the 3.0.5 SDK, which I renamed *GradientImport*, which was more in sync with what it did.

7.0 GradientImport import plug-in module

So you thought the Dissolve example was torture enough? Oh no, things get much more fun when you try to apply scripting to a module that can be controlled in a *batch*. *Batch importing* is an additional method for loading numerous images at a time. This is in addition to the old *multiple acquire* mechanism that is part of the import module interface.

The batch command is available from the pull-down menu attached to the actions palette.

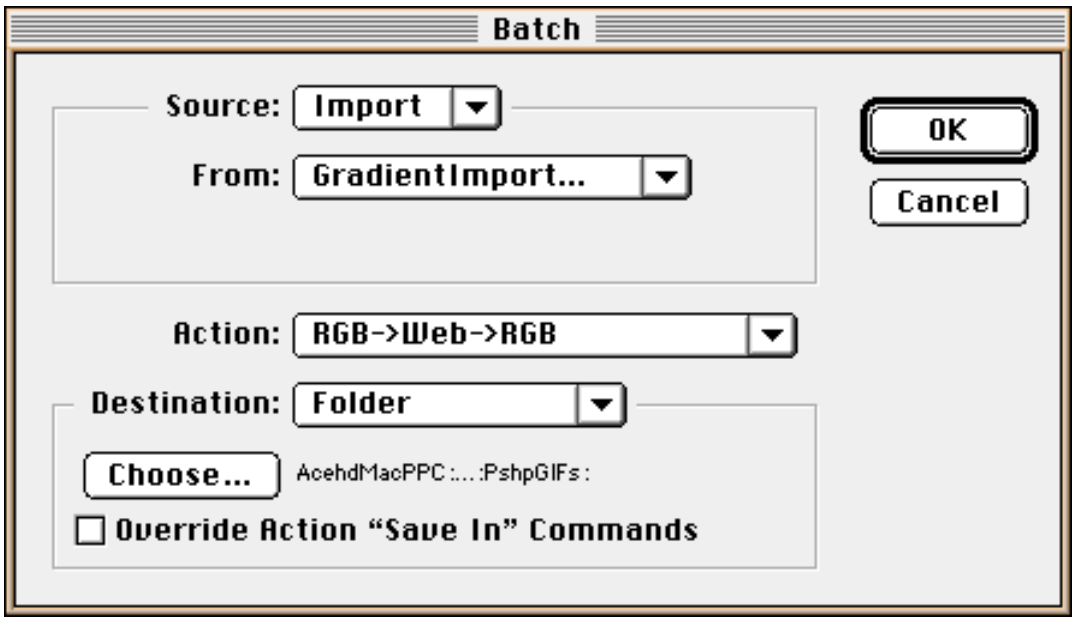

Figure 6: Batch dialog

With so many options, there are several approaches to updating an Import module:

1. Leave it alone. The scripting system will automatically call the import module for each import in a batch. Even vanilla plug-ins can be called by the scripting system. Your dialog will be popped for every iteration, which may not be desireable.

2. If it is a single import module, meaning it only returns one image at a time, you can update it for scripting and record all the parameters necessary for that single import. The batch mechanism will pass your parameters to your plug-in automatically.

3. If it is a multiple acquire module, that means that all control for opening multiple images happens within your plug-in. You can: a) maintain detailed control over the iterative imports and use the scripting system to call your plug-in with some default parameters, such as preferences, and/or b) record every iterative import as another scripting event.

The GradientImport module uses the older multiple acquire mechanism. To showcase the most robust scripting setup, I chose the last option, 3b, and decided to make the plug-in record every event of its multiple acquire. That way a user can blast off a single action and have multiple images open. This makes the most sense for digital cameras that cache a set of images and let the user import and color correct multiple images.

7.1 Creating the GradientImport terminology resource

7.1.1 Assessing the user interface

The first thing I did was examine the user interface dialog to determine what parameters to represent in the terminology resource.

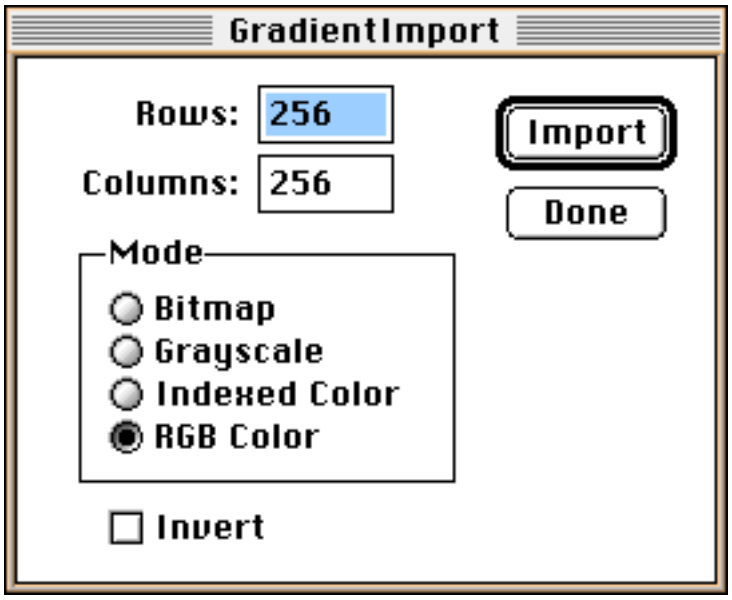

Figure 7: GradientImport user interface

The items were:

1. An “OK” button (“Import”) which does not need to be recordable.
2. A “Cancel” button (“Done”) which does not need to be recordable.
3. An integer from 1 to 30,000 representing Rows
4. An integer from 1 to 30,000 representing Columns
5. A mutually-exclusive enumeration, “Mode”, representing “Bitmap”, “Grayscale”, “Indexed Color”, or “RGB Color”.
6. A boolean, “Invert”

Below is the terminology resource I used for GradientImport.

7.1.2 GradientImport terminology resource

```
resource 'aete' (ResourceID, purgeable)
```

```
{  // aete version and language specifiers:
   1, 0, english, roman,
   {  // vendor suite name:
      vendorName,                          // "AdobeSDK"'
      "Adobe example plug-ins",            // optional description
      ourSuiteID,                          // suite id 'sdK3'
      1,                                   // suite code, must be 1
      1,                                   // suite level, must be 1
      {},  // structure for filters
      {  // structure for all other plug-in types:
         vendorName " GradientImport",     // "AdobeSDK GradientImport"
         "gradientImport multiple import",// optional description
         {  // properties:
            "<Inheritance>",
            /* all non-filters inherit from a base class of the same name as their plug-in
            type, such as classFormat, classExport, etc. See PIActions.h. Inheritance must
            be the first property entry, even if there are no others. */
            keyInherits,                   // always
            classImport,                   // classExport, classFormat, etc.
            "parent class import",         // optional description
            flagsSingleProperty,           // parameter flags

            // Second property:
            "multi-import",                // property name
            keyMultiImportInfo,            // unique key 'mulK'
            classMultiImportStruct,        // unique class 'mulS'
            "multiple import info",        // optional description
            flagsListProperty              // flags for a list
         },  // close properties
         {},  // elements (not supported)
         /* Normally you won't need to create other classes, but since I'm going to be
         storing a list of "import information" (the values needed to create one image),
         I'm creating a class with the set of information, called "import info": */
         "import info",                    // unique class name
         classMultiImportStruct,           // unique class 'mulS'
         "class import info",              // optional description
         {  // import info class properties:
            "rows",                        // property name
            keyRows,                       // standard key keyHorizontal
            typeFloat,                     // property type
            "number of rows",              // optional description
            flagsSingleProperty,           // flags for property

            "columns",                     // property name
```

```
            keyColumns,                     // standard key keyVertical
            typeFloat,                      // property type
            "number of columns",            // optional description
            flagsSingleProperty,            // flags for property

            "mode",                         // property name
            keyOurMode,                     // standard key keyMode‘
            typeGradientMode,               // unique type ‘grmT’
            "color mode",                   // optional description
            flagsEnumeratedProperty,        // flags for property

            "invert",                       // property name
            keyInvert,                      // unique key ‘invR’
            typeBoolean,                    // property type
            "invert image",                 // optional description
            flagsSingleProperty             // flags for property
         } // close class import info
         {},  // elements (not supported)
      } // close non-filter classes
      {},  // comparison operators (not supported)
      { // Any enumerations go here. We have one, typeGradientMode:
         typeGradientMode,                  // unique type ‘grmT’
         { // enumeration listing:
            "bitmap",                       // property name
            ourBitmapMode,                  // unique key ‘bitM’
            "bitmap mode",                  // optional description

            "grayscale",                    // property name
            ourGrayscaleMode,               // unique key ‘gryS’
            "grayscale mode",               // optional description

            "indexed color",                // property name
            ourIndexedColorMode,            // unique key ‘indX’
            "indexed color mode",           // optional description

            "rgb color",                    // property name
            ourRGBColorMode,                // unique key ‘rgbC’
            "rgb colormode",                // optional description
         }, // close typeGradientMode
      } // close enumerations
   } // close vendor suite
}; // close ‘aete’
```

After the terminology resource was done, I added the HasTerminology to the PiPL.

7.1.3 GradientImport HasTerminology PiPL property

```
HasTerminology { ourClassID, ourEventID, ResourceID, uniqueString }
```

With:

```
#define vendorName    "AdobeSDK"          // unique vendor name
#define ourSuiteID    'sdK3'              // must follow id guidelines
#define ourClassID    'graD'              // must be unique, but can be suite id
#define ourEventID    typeNull
/* must be typeNull or the host will think it's a filter (event) instead of an import, export,
format, or selection (class) */
#define ResourceID    16000               // typical id for plug-ins
#define uniqueString  ""                  // empty
```

7.2 Writing scripting parameters in GradientImport

The next step was to create the routine to pass the scripting parameters back out to Photoshop. Taking the same approach as with the Dissolve example, I used my globals to pass their values across my different functions, then, at the last minute, I pass the list of events back encapsulated in a descriptor.

Due to the nature of the multiple acquire mechanism, I needed a way to track the multiple imports that would occur and then hand them back to the scripting system. I decided to do this by creating an actual descriptor for each import, then storing all the descriptors inside an encapsulating descriptor to hand back to the host at the very end of execution. This took the form of:

1. In `DoFinish`, create a descriptor and store it in a static array with a maximum of `kMaxDescriptors` (in this case, 50) via `CreateDescriptor()`.

2. In `DoFinish`, if multiple acquiring was not available, write the descriptor out to the host in final form via `CheckAndWriteScriptParams()`.

3. In `DoFinalize`, write the descriptor out to the host in final form via `CheckAndWriteScriptParams()`.

So, DoFinish looked like this:

```
void DoFinish (GPtr globals)
{
   gStuff->acquireAgain = gContinueImport;
   // gContinueImport tracks whether to continue importing

   // Now create a descriptor and store it in our static array for saving later:
   CreateDescriptor(globals); // creates and stores descriptor in next open gArray

   // If we can't finalize, then we'll have to write our parameters now:
   if (!gStuff->canFinalize)
      CheckAndWriteScriptParams(globals); // writes script params
}
```

And DoFinalize:

```
void DoFinalize (GPtr globals)
{
   gQueryForParameters = false;  // reset global
   CloseOurDialog (globals);  // closes our UI

   // We're done. Write final parameters:
   CheckAndWriteScriptParams(globals); // writes script params

}
```

I created a source file, GradientImportScripting.c, where I put all the scripting routines.

```
void CreateDescriptor (GPtr globals)
{
   PIType               mode = GetGradientMode(gLastMode);
   // converts a global enumeration to the actual unsigned32 mode

   const double         rows = gLastRows, columns = gLastCols;
   // converting globals to doubles for PutUnitFloat to use unitPixels value

   Boolean              invert = gLastInvert;
   PIWriteDescriptor    token = NULL;
   PIDescriptorHandle   h;
   OSErr                stickyError = noErr;

   if (DescriptorAvailable())
   { // PIUtilities routine to check for descriptorParameters callbacks succeeded.
      token = OpenWriter(); // open new write descriptor
      if (token)
      {  // got the descriptor. Go ahead and write the keys into it:
         PIPutUnitFloat(token, keyRows, unitPixels, &rows);
         // puts our rows as pixels
```

```
            PIPutUnitFloat(token, keyColumns, unitPixels, &columns);
            // puts our columns as pixels

            PIPutEnum(token, keyOurMode, typeGradientMode, mode);
            // puts the exact enumeration (must match terminology resource!)

            if (invert) PIPutBool(token, keyInvert, invert);
            // again, only if non-default (true), writes "with invert"

            stickyError = CloseWriteDesc(token, &h);
            /* have to call PIUtilities CloseWriteDesc, which closes a specific token, and
            returns a descriptor handle in "h". If I called CloseWriter, it would close it and
            automatically store it in gStuff–>descriptorParameters, which I don't want, since
            I'm trying to create a static array of descriptors before passing them to the host. */
            token = NULL; // just in case

            if (!stickyError)
            { // as long as we didn't have an error writing:
               if (gLastImages >= kMaxDescriptors)
               { // oops, went over our limit. Delete the last and replace it:
                  gLastImages--; // just keep replacing last one
                  PIDisposeHandle(gArray[gLastImages]);
                  // dispose last handle
               }

               gArray [gLastImages++] = h; // stick handle on array

               gArray [gLastImages] = h = NULL; // null out end, just in case}
            } // close stickyError
         } // close token
      } // close descriptorAvailable
} // end createDescriptor
```

The `CheckAndWriteScriptParams` routine checks for any data then calls the `WriteScriptParams` routine:

```
OSErr CheckAndWriteScriptParams (GPtr globals)
{
   OSErr                gotErr = noErr;

   if (gLastImages) gotErr = WriteScriptParams(globals);
   // if we have done at least one import (gLastImages > 0), write our scripting parameters
   else gotErr = gResult = userCanceledErr;
   /* else error out of entire loop (if we don't do this, we might end up with a single recorded
   parameter, "Import using: GradientImport" which looks ugly. */
   return gotErr;
}
```

```
OSErr WriteScriptParams (GPtr globals)
{
   unsigned32           count = gLastImages;
   PIWriteDescriptor    token = NULL;
   OSErr                stickyError = noErr;

   if (DescriptorAvailable())
   { // gStuff–>descriptorParameters callbacks available.
      token = OpenWriter(); // open write descriptor
      if (token)
      { // got our token. Write our keys.
         PIPutCount(token, keyMultiImportCount, count);
         /* A list is always preceded by its count. Note the count, and the following keys,
         are stored as keyMultiImportCount for the entire list. */

         for (count = 0; count < gLastImages; count++)
         { // iterate through local array:
            PIPutObj(token, keyMultiImportInfo,
            classMultiImportStruct, &gArray [count]);
            /* PIPutObj, from PIUtilities, automatically disposes the handle and sets it to
            null. */
         }

         gLastImages = 0; // reset
         stickyError = CloseWriter(&token);
         /* closes descriptor, stores it in gStuff–>descriptorParameters, sets
         plugInDialogOptional, and sets token to null. */
      } // close token
   } // close descriptorAvailable
   return stickyError;
} // end WriteScriptParams
```

7.3 Testing the multiple import routine

Now that the write routines are done, I was able to test the multiple import routines. I turned recording on in the actions palette and imported a couple of images, one after the other, then dismissed the GradientImport dialog. Figure 8 shows the resulting display in the actions palette.

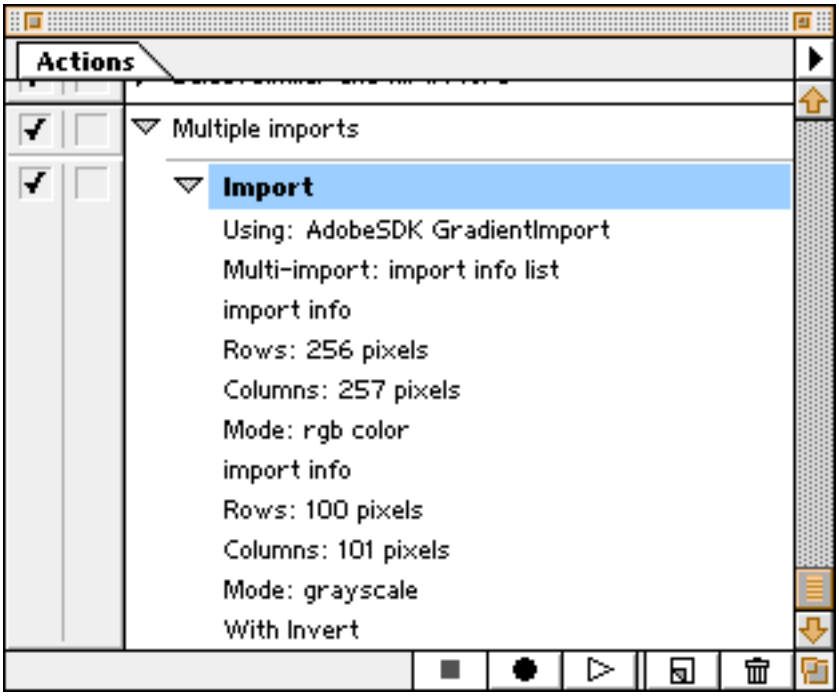

Figure 8: GradientImport display in the actions palette

Note how the multiple import list is presented: as its label, “Multi-import”, with its type label, “import info” and “list” after it. Then each individual item of the list is headed with the type label “import info”. The first image is a 256x257 RGB image; the second image is a 100x101 grayscale inverted image. Again, I only display a boolean when its in its non-default (“with invert” only, as opposed to “without invert” and “with invert”). Another nice feature is the display of the word “pixels” after the “Rows” and “Columns” entries. This is thanks to `PutUnitFloat` and `unitPixels`.

7.4 Playback of scripting parameters for GradientImport

Now that I had GradientImport correctly recording parameters, it was time to modify it to read back parameters. This, too, is complicated, because it requires reading from a list and dispatch parameters through the multiple acquire loop, iterating through the list. I decided to break it out into this logic:

1. At `DoPrepare`, open any descriptor handed to me by the host and see if there was a list in there, via `OpenScriptParams`.

2. At `DoStart`, read the next descriptor object in the list via `ReadScriptParams` and assign all its keys to globals via `SwitchScriptInfo`

3. In `DoStart`, as soon as the dialog is asked for, or if there is an error, we no longer need to iterate through the list. Close it via `CloseScriptParams` and continue to create our own array to pass back later.

```
void DoPrepare (GPtr globals)
{
   gStuff->maxData = 0;

   if (!WarnBufferProcsAvailable ())
      gResult = userCanceledErr; // exit. Already displayed alert.

   // if finalization is available, we will want it:
   gStuff->wantFinalize = true;

   ValidateParameters (globals);
   /* this should look familiar. Same functionality, but instead, checks variables pertinent to
   GradientImport for default values and allocation, if needed. */

   // now see if the scripting system has passed us anything:
   OpenScriptParams (globals);
}

void DoStart (GPtr globals)
{
   int16 j = 0;

   // Insist on having the buffer procs:
   if (!WarnBufferProcsAvailable ())
   {
      gResult = userCanceledErr; // should probably display err
      return;
   }

   // Assume we won't be coming back around for another pass unless explicitly set:
   gStuff->acquireAgain = gContinueImport = false;

   // Validate our globals then override them with scripting parameters, if available:
   ValidateParameters (globals);
   ReadScriptParams (globals);

   if (gQueryForParameters)
   { // open our dialog. If it's already up, this returns with no err:
      if (!OpenOurDialog (globals))
      { // Couldn't open our dialog. Abort! Abort!
         gQueryForParameters = false;
         CloseScriptParams(globals); // Close up the open descriptor!
         gResult = memFullErr; // return with memory full error
         return;
      }
```

```
// So far so good. Now dispatch our dialog routines:
if (!RunOurDialog (globals))
{ // User canceled. Close everything up.
   gQueryForParameters = false;
   CloseOurDialog (globals); // deallocates dialog
   CloseScriptParams(globals); // closes open descriptor
   gResult = userCanceledErr; // exit without err
   return;

   // rest of DoStart here.
```

With `DoPrepare` and `DoStart` set up, there were four routines to be created. `OpenScriptParams`, to open the descriptor; `ReadScriptParams`, to read the next object in our list; `SwitchScriptInfo`, which reads keys from the object and overrides the global values, and `CloseScriptParams`, to close and tidy up the open descriptor handed to the plug-in from Photoshop.

`OpenScriptParams` was one of the easier ones, as all it had to do was watch for the count key and find it in the descriptor handed in by the host:

```
void OpenScriptParams (GPtr globals)
{
   DescriptorKeyID      key = 0;
   DescriptorTypeID     type = 0;
   int16                loop = 0;
   int32                flags = 0;
   Boolean              leaveEarly = false;

   if (DescriptorAvailable())
   { // descriptor procs available. Now open the descriptor:
      gToken = OpenReader(NULL);
      /* Normally would pass an array indicating the expected keys. Problem is I don't
      know how many items are in the list until I open it. Therefore, I'm passing NULL to
      indicate to the scripting system not to bother with a key array list. */
      if (gToken)
      { /* since we'll be reading from this descriptor in numerous routines, I store the
      token in a global variable. */
         while (!leaveEarly)
         { // Until we find our key or run out of keys in the descriptor, we'll look for it:
            leaveEarly = PIGetKey(gToken, &key, &type, &flags);
            switch (key)
            { // Only interested in one case, keyMultiImportCount:
               case keyMultiImportCount:
                  PIGetCount(gToken, &(gCount));
                  leaveEarly = true;
                  break;
               /* I'm ignoring all other keys. All I'm looking for is the list, which will be
               preceded by a count key. Once I find that, I drop out, eventually to be
               called by the read routine. */
            } // close switch
         } // close leaveEarly
```

```
        } // close gToken
        gQueryForParameters = PlayDialog();
        // if true, show the dialog

    } // close descriptorAvailable
} // end OpenScriptParams
```

The `ReadScriptParams` routine needs to take up where the `OpenScriptParams` routine left off: There is an open descriptor, `gToken`, and it is sitting on an object which is another descriptor. I need to take that descriptor, open it, parse all its keys, and override my globals. That happens in `SwitchScriptInfo`.

```
void ReadScriptParams (GPtr globals)
{
    int16                loop = 0;
    int32                flags = 0;
    DescriptorTypeID     type = 0;
    DescriptorKeyID      key = 0;
    PIDescriptorHandle   subHandle = NULL;
    PIReadDescriptor     subToken = NULL;
    OSErr                stickyError = noErr;

    DescriptorTypeID     passType = classMultiImportStruct;
    // GetObj needs to know what class type to expect

    DescriptorKeyIDArray subKeyIDArray =
    { keyRows, keyColumns, keyOurMode, NULLID };
    /* These are all expected. If keyInvert is there, it's handled, just not checked off the list. If I
    put it in the list, then the list will generally always return with an error, saying it didn't
    get keyInvert. I'd rather have it be a pleasant addition then always expecting it and rarely
    getting it. */

    if (DescriptorAvailable())
    { // Have descriptor procs.
        if (gToken)
        { // global token is valid
            if (gCount > 0)
            { // have another item waiting
                gLastInvert = false;
                /* default is no invert. If we get the key, we'll override the default. Otherwise,
                we set it here, just in case we have an error below and don't get a chance to set
                it one way or the other. */

                PIGetObj(gToken, &passType, &subHandle);
                /* From PIUtilities, reads an object from descriptor gToken into subHandle of
                type passType */

                subToken = OpenReadDesc(subHandle, subKeyIDArray);
                /* Can't use OpenReader() because that automatically uses the descriptor
                passed in gStuff–>descriptorParameters. Instead, we use a subroutine,
```

```
                OpenReadDesc, which opens handle subHandle and tracks array
                subKeyIDArray, returning its descriptor token. */
                if (subToken)
                { // was able to open descriptor.
                   SwitchScriptInfo (globals, subToken);
                   // reads the keys from descriptor subToken and overrides globals

                   stickyError = CloseReadDesc(subToken); // done
                   subToken = NULL; // just in case
                   PIDisposeHandle(subHandle); // dispose handle
                   subHandle = NULL; // just in case

                   if (stickyError)
                   { // error occurred while reading keys
                      if (stickyError == errMissingParameter)
                         ; /* -1715 missing parameter. Walk keyIDArray to find which
                         one. */
                      else
                         gResult = stickyError; // real error occurred
                   }
                   gContinueImport = true; // we got something, so keep going!
                } // close subToken
                gCount--; // one less in list
             } // close count
             if (gCount < 1)
                CloseScriptParams(globals); // that was the last one! Close it up!
          } // close readToken
       } // close descriptorAvailable
    } // end ReadScriptParams
```

The `SwitchScriptInfo` routine reads keys out of the descriptor, overriding their global values:

```
void SwitchScriptInfo (GPtr globals, PIReadDescriptor token)
{
   DescriptorKeyID      key = 0;
   DescriptorTypeID     type = 0;
   int16                loop = 0;
   int32                flags = 0;
   int32                count = 0;

   double               rows = kRowsMin, columns = kColumnsMin;
   // default value for rows and columns
   PIType               mode = ourRGBColorMode;
   // default value for mode is RGB
   Boolean              invert = false;
   // default for invert is false
```

```
    const double          minRows = kRowsMin, maxRows = kRowsMax,
                          minColumns = kColumnsMin,
                          maxColumns = kColumnsMax;
    /* PinUnitFloat will pin a value between minimum and maximum bounds, but, since
    those values are passed as addresses, I assign these locals to the constant values */

    unsigned long         pixelsUnitPass = unitPixels;
    // have to pass address of unsigned long for unitPixels, so assign local to constant

    while (PIGetKey(token, &key, &type, &flags))
    { // continue while there are more keys
       switch (key)
       {
          case keyRows:
             PIGetPinUnitFloat(token, &minRows, &maxRows,
             &pixelsUnitPass, &rows);
             /* pins the value between min and max, returnning it in "rows". It will return
             coercedParam if it had to coerce the value to between min and max */
             gLastRows = rows; // assign local double to global short
             break;
          case keyColumns:
             PIGetPinUnitFloat(token, &minColumns, &maxColumns,
             &pixelsUnitPass, &columns);
             // pins columns between min and max
             gLastCols = columns; // assign local double to global short
             break;
          case keyOurMode:
             PIGetEnum(token, &mode);
             // returns an enum -- must be the same as terminology enum list
             gLastMode = GetPlugInMode(mode);
             // maps enum to ordinal
             break;
          case keyInvert:
             PIGetBool(token, &invert); // returns boolean
             gLastInvert = invert; // assigns boolean to global
             break;
       } // close switch
    } // close getkey
} // end SwitchScriptInfo
```

`CloseScriptParams` is called from multiple places whenever there is an error or the list is finished and the descriptor passed to the plug-in by Photoshop should be closed. Note that the descriptor passed by the host is a handle, and is the plug-in's responsibility to deallocate. If I didn't call this routine, we'd have a memory leak, unless I passed the exact same descriptor back to the host. But I don't pass the same descriptor back, because, even while this open descriptor is being read and used to do multiple imports, etc., the `CreateDescriptor`, etc., routines are creating descriptors to pass back to the host in `WriteScriptParams`. Ergo, since I'm putting my own descriptor in `gStuff->descriptorParameters`, I have to call

CloseScriptParams, at least once, to make sure that the host descriptor is disposed.

```
void CloseScriptParams (GPtr globals)
{
   OSErr                stickyError = noErr;

   if (DescriptorAvailable())
   { // descriptor procs available
      if (gToken)
      { // have our global token
         stickyError = CloseReader(&gToken);
         // closes token, deallocates memory, and sets it to null

         if (stickyError)
         { // oops, got an error
            if (stickyError == errMissingParameter)
               ; // -1715 missing parameter. Sort of late, by now.
            else
               gResult = stickyError; // real error occurred
         }
      } // close token
   } // close descriptorAvailable
   gCount = 0; // reset global list count
   gContinueImport = false; // finish importing and exit
} // end CloseScriptParams
```

7.5 Playing back GradientImport

Now that the playback functions have been completed, the last task was to record some actions and play them back to make sure the parameters were honored. It's pretty cool to create a single action that contains multiple imports inside of it, and you can see how the actions palette can get pretty full.

8.0 Other issues and future implementation

8.1 Opaque data

You can see that the actions palette can fill up pretty fast with large multiple imports. *Opaque data* is the term for information that you don't want displayed in the actions palette. This is sometimes useful because the data is serial or registration information, it's complex, cannot be represented to the user given the current interface (the actions palette), or simply looks yucky.

In PIActions.h there is a key, "keyDatum" (I couldn't use keyData, it was taken) that displays in the actions palette as:

Data: "..."

Which is an opaque display. keyDatum (and other opaque keys) must be stored as *textual* data. That means that if you want to store an array of hexadecimal values, for instance, you must convert them to their textual representation. To store:

```
$01 $02 $03 $04 $05
```

You must store it as:

```
"0102030405"
```

Or some such similar representation. The reason for this, and the reason there are no opaque keys that simply do not display at all in the actions palette, stems from the user interface issues of the AppleScript and AppleEvent automation architecture. Without getting into too much detail, it has to do with the fact that the user side of the architecture is made so that a user may pass any English-like string into the automation system to be parsed, such as:

```
tell application "Photoshop" to do Gaussian Blur with radius 2.0
```

Opaque data breaks this mold, but not completely, because opaque data, by its definition, has no English equivalent. (Otherwise, you would just display it in the actions palette like any other parameter.) Because strings and sentences can be passed as automation and event requests, even the opaque data must be able to be typed and passed as a simple sentence. So, by this example, the user could pass the event:

```
tell application "Adobe Photoshop 4.0" to do GradientImport with data
"0102030405"
```

There is more detail on this in the AppleScript and AppleEvent *Inside Macintosh* books, and references to them in the Photoshop SDK Guide.

8.2 External scripting

Scripts can be controlled via OLE on Windows and AppleScript on Macintosh. Documentation on triggering scripts externally is in the *Photoshop SDK Guide* in the Scripting chapter and in *Appendix B: OLE Automation*.

8.3 Saving filenames

What isn't covered in the scope of this article, but is an interesting scripting question, is what to do with filenames when saving them as scripting keys. I recommend looking at the example Format Module in the SDK for an example of this. The basic logic used by Photoshop for converting the filename dialog into a scripting parameter, and, therefore, the logic I recommend you use is:

1. If the user types a new name, save that entire path.

2. If the user leaves the default name, save the path to the folder, but append the current filename to the path when saving.

More detail about this is can be found in the SDK guide and the Format example.

8.4 Future features

Photoshop 4.0 scripting is available to all plug-in module types, and, as stated, it can control non-scripting aware plug-ins by executing them as if a user had selected them.

We recommend that you update your plug-in to be Photoshop 4.0 scripting-aware. Because execute-only plug-ins pop their user interface every time they're called from an action, a user running a batch on a folder of hundreds of files is going to have a much more positive experience, and therefore prefer, working with plug-ins that have been made scripting-aware.

I recommend playing with the batch control mechanism to get a good understanding of how it interacts with the user, and also to look at how Save and Open dialogs are handled, as far as scripting is concerned.

Next issue I'll take a look at some of the new plug-in types introduced in Photoshop 4.0 and all the new API features related to those, including color picker plug-ins and the new selection modules.

###