Programming for e-Learning Developers

ToolBook®, Flash®, JavaScript™, and Silverlight™

By

Jeffrey M. Rhodes

Platte Canyon Press
Colorado Springs

Creating an Interactive "Rollover" Screen

Let's make an interactive screen where the names of Beatles albums are listed down the left side. When the user rolls her mouse over the names, they change color and a graphic of the album cover is displayed on the right side of the screen. We'll also include a track list from www.beatles.com . Album names that have been "rolled over" will be shown in a third color so that the user can tell which ones she has already completed. Finally, we will display a message when all of the rollovers have been completed. This is a pretty realistic interactive training screen and will introduce us to a number of important concepts.

ToolBook – OpenScript

You might take a look at Figure 38 so that you can picture our task. I assembled the graphics as .png files and created fields along the left with the titles of the albums. The names of these fields are "album 1" through "album 5."[1] I imported the graphics as *resources*[2] and gave them matching names. Finally, I went to beatles.com and grabbed the track listing for each album and put them in fields named "field 1" through "field 5." I hid these fields as we are going to read their text (technically their *richText* so that we preserve any formatting) and show them in a common "display field." I like this technique because it avoids the need to reposition all the fields if we show and hide them in turn. Our design is then that we'll show the album cover in a button and set this display field to be the track listing in response to the *mouseEnter* event, which is what ToolBook calls a rollover.

[1] You might be wondering why I put a space between the base (album) and the number. Unlike some of the other environments, ToolBook allows this. The advantage comes when it is time to "parse" the number from the name. With OpenScript, we can use syntax like this: *word 2 of name of target*. The space between the base and the number is what separates word 1 from word 2. This keeps us from running into problems when we get to *album 11*. In environments that don't allow spaces in object names, I'll put an _ instead and use some kind of *split* method for the same reason. You will see that in later examples.

[2] Similar to putting them into the Library in Flash. Silverlight has both *content* (inside the .xap file) and *resources* (inside the DLL itself).

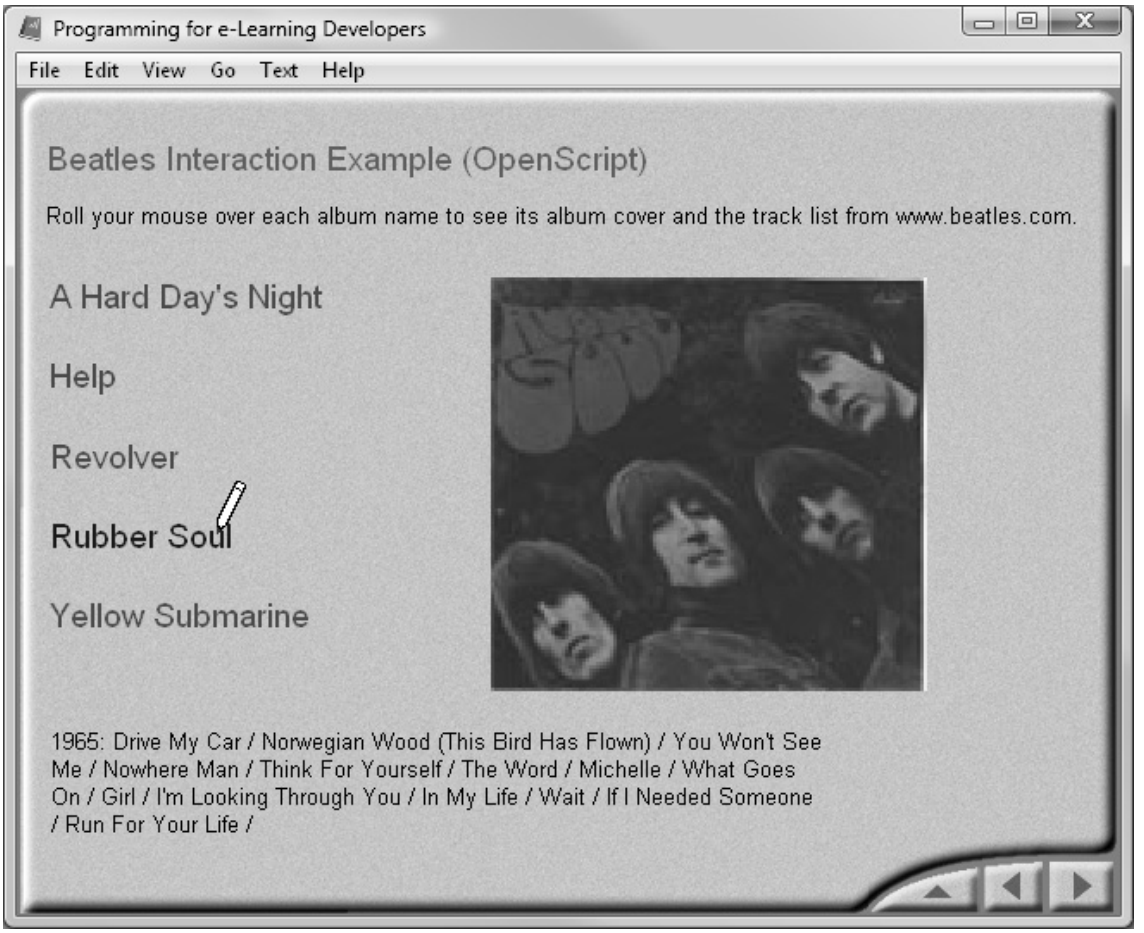

Figure 38. Interactive Example - ToolBook OpenScript.

We are now ready to do some programming. Let's start with the *mouseEnter* script shown in Listing 18. We put this script at the page level so that we can write one mouseEnter handler for all five album name fields.

```
to handle mouseEnter
 system lastFieldId
 system stack completedInteractionList
 system dword numInteractions
 local string tarName
 local string tarNum
 local field fieldId

 tarName = name of target
```

```
 if word 1 of tarName = "album"
  tarNum = word 2 of tarName
  fieldId = field "display field"
  richText of fieldId = richText of field ("field " & tarNum)
  normalGraphic of button "albumImage" = bitmap ("album " & tarNum)
  strokeColor of target = blue
  sysCursor = 19 -- pen
  if lastFieldId <> null
   strokeColor of lastFieldId = 120,25.125,100 -- dark green
  end if
  lastFieldId = target
  -- check for completion
  if ASYM_ItemInList(tarNum, completedInteractionList) = False
   push tarNum onto completedInteractionList
  end if
  if itemCount(completedInteractionList) >= numInteractions
   richText of fieldId = "COMPLETED: " & richText of fieldId
  end if
 end if
 forward
end mouseEnter
```

Listing 18. Interactive Example Implementation (mouseEnter) - ToolBook OpenScript.

We have lots of good programming concepts to discuss in this script. The first is a *global* variable, which we might define as some data that needs to survive beyond the life of the current handler or function. In OpenScript, we declare a global variable with the word *system* in front of it. In most cases, we want to declare a type as well. We do this so that the environment will help us if we do something dumb like try to assign “Joe” to a variable that is supposed to be a number.[3] So the line *system dword numInteractions* means a global variable of type *dword*[4] of the name *numInteractions*[5]. In the mouseEnter script, we need three global variables:

1. A reference to the “previous” field that we entered. To understand this, we need to think through the set of events. The user will move his mouse into “Rubber Soul.” At that point, we want to turn it blue. He then moves the mouse into “Help.” We then turn “Help” blue to denote that it is the current item. We want to turn “Rubber Soul” green to show that we have already visited it. To do that, we need to remember which field we were in last. That is why we have *lastFieldId*. We don’t declare a datatype in this case because OpenScript is flexible enough to make it a *field* reference when we are using it but then allow us to set it to *null* when

[3] There are numerous references and studies about the cost of fixing bugs across the lifecycle. As you might expect, it is far easier and cheaper to fix bugs before your e-Learning leaves your desk than to get user reports, find the right files, fix the problem, and re-deploy. Declaring data types is an important first step in reducing bugs.

[4] A dword in ToolBook is a 32-bit unsigned integer, meaning that it can only be positive, whole number.

[5] Notice how we name the variable with the first letter lowercase and the first letter of each word within the variable name as uppercase. This is called “camel casing” (since each new word looks like a hump) and is the recommended naming convention.

entering the page (Listing 19). If we type the variable, ToolBook would give us an error when we try to set it to null.

2. In addition to knowing the most recent field that the user entered, we need to keep track of all the fields in order to determine when the user has completed all of them. There are a number of ways we could approach this, but one of the simplest is to have a comma-delimited list of completed interactions (1,3,5 for example). This is a *stack* data type that we store in the *completedInteractionList* variable. It again needs to be global since we need to build up this stack interaction by interaction.
3. Finally, we need to know how many interactions there are in order to figure out if we are finished. This doesn't strictly need to be a global variable but we end up using this value in two different handlers (*mouseEnter* and *enterPage*). The advantage of a global variable here is that we only have to change the value once if we change the number of interactions.[6]

After the global variables, we have three *local* variables. This means that they survive only until the end of the handler or function. The *tarName* variable allows us to avoid having to keep referring to *name of target*. Similarly, we end up grabbing the interaction number and putting it in the *tarNum* variable[7]. Finally, we end up referring to our display field several times. It is a good practice to put this object reference in a variable, which we call *fieldId*. This is a bit more efficient for the program if it doesn't need to keep "resolving" the object reference and gives us a little less code to write.

Let's now look at the logic. We use an *if* statement to limit our logic only to targets that have as their first word "album." We need to do this since every object on the page will generate the *mouseEnter* event. Next, we populate our *tarNum* variable with word 2 of the name (1, 2, etc.). We build our *fieldId* reference to the display field and then start on the cooler stuff. We set the *richText* property of the display field to the *richText* of the corresponding (hidden) field that holds the album information. We use *richText* instead of *text* since the former keeps all the formatting such as font size, bold, and italics. After that we set the *normalGraphic* property[8] of our "albumImage" button to the corresponding bitmap resource. Notice how we use "dynamic object referencing" to move from the *name* of the object to the field or bitmap object itself. The next line sets the *strokeColor* (the color of the text) to blue. We then set the *sysCursor* property[9] to one of its defined values, which corresponds to a graphic that looks like a pen. I like to change the cursor for a mouseEnter interaction as another visual clue to the user that something is happening.

---

[6] In "real life," I would actually use a *user property* instead, but we haven't covered those yet. The reason is that ToolBook global variables are truly global for the entire book, so setting up this *numInteractions* variable will make it persist (and take up memory) for all the other pages. In Flash, JavaScript, and Silverlight, you can declare variables outside a function block that can be shared between the handlers but are not persisted to the rest of the program.

[7] You might be wondering why we type this as a string instead of a number. Either would work but technically we get the interaction number by pulling the last part off the name of the target and it is therefore a string. We then use that to build the name (again a string) of the album graphic and information fields. If we were in a stricter environment like Visual Basic, we would need to convert back and forth between a number and string (similar to how we had to use *CType()* in Listing 17). It is simpler just to leave it a string.

[8] The "normal" comes from the different button states (invert, checked, etc.) that can each have a graphic associated with it.

[9] It is a bit unusual in that this is not a property of a particular object. This is basically a property of the "application."

We now use our *lastFieldId* global variable discussed above. We check to see if it is *null* (because it won't be defined yet the first time). If not, we set its *strokeColor* to a dark green. Either way, we set this global variable to the *target* (e.g., the current field the user is in). That way, we'll set this field to green during the next interaction.

Our last task is to check for completion. We previously defined the *completedInteractionList* global variable and explained our plan to use it as a comma-delimited list of the interactions the user has completed. One reason to choose this format is that OpenScript has excellent support for "stacks" like this. We first use the built-in *ASYM_ItemInList()* method to check if the current *tarNum* is in our global variable. If not, we *push* it on the variable, which has the effect of adding both the value and the comma (once there are two or more entries).[10] We then use another built-in method, *itemCount*, to see if the number of items in the list is greater than or equal to[11] to our *numInteractions* global variable. If so, we update our display field to show "COMPLETED: " at the front. In a real e-Learning application, I would typically change the look of the "Next Page" button or show an animation, but we already have enough complexity in this example!. We end by forwarding the *mouseEnter* message so that higher-level scripts can handle the message if desired.

That script had most of the heavy lifting. Listing 19 has the "initialization" and "cleanup" scripts.

```
to handle enterPage
 system lastFieldId
 system dword numInteractions
 system stack completedInteractionList

 numInteractions = 5

 step num from 1 to numInteractions
  strokeColor of field ("album " & num) = 20,30,100 -- dark orange
 end step
 text of field "display field" = ""
 clear normalGraphic of button "albumImage"
 lastFieldId = null
 completedInteractionList = ""
 forward
end enterPage

to handle mouseLeave
 local string tarName
```

[10] We want to use *ASYM_ItemInList* rather than a method like *contains* because we don't want to set ourselves up for problems between 1, 10, 11, etc. If we have completed interaction 10 but not interaction 1, *ASYM_ItemInList()* will return False when looking for "1" in the list but *contains* will return True.

[11] You might be thinking that there is no way that there can be more items in the list than the number of interactions. You are right but this is another example of defensive programming. If this *did* happen somehow (such as if we added another field but did not change our *numInteractions* global variable), we'd want to go ahead and mark the page as completed.

```
  tarName = name of target

  if word 1 of tarName = "album"
   sysCursor = default
  end if
  forward
end mouseLeave
```

Listing 19. Interactive Example Implementation (enterPage and mouseLeave) - ToolBook OpenScript.

The way to look at the *enterPage* handler is that we want to initialize the page to our desired state.[12] We need to set our global variables[13] so they are defined at the top of the script. We initialize *numInteractions* and then use our first *step* loop[14] to set the *strokeColor* of each of our fields to a dark orange. In a step loop, the variable (*num*) goes from the initial condition (*1*) to the final value (*numInteractions)*. The code within the loop runs each time. Notice how many lines of code we save plus get more flexibility to change the number of interactions without adding code by building the name of the field (*"album " & num*), letting OpenScript build a reference, and setting the color. From there, we clear our display field and "albumImage" button. Finally, we clear our *lastFieldId* and *completedInteractionList* variables. It is very important that we forward the *enterPage* message - lots of other things happen in ToolBook at higher levels in response to this message.

The *mouseLeave* handler is quite simple. We use the same logic as in Listing 18 to make sure we are only handling the event for our album name fields. If so, we set the *sysCursor* back to default.

That certainly took much longer to explain than to actually program. But the remaining examples should go faster as the logic will be similar.

ToolBook – Actions Editor

Accomplishing our task in the ToolBook Actions Editor takes a little longer due to the lack of dynamic object referencing, but the outcome is still quite functional. The end result is shown in Figure 39. The main change from the OpenScript solution is that we need to create individual buttons showing the album covers. We have named these "albumGraphic 1" to "albumGraphic 5." We also scrap the changing of the cursor as the Actions Editor doesn't support that[15].

[12] In a native ToolBook application, you will typically do this on the *leavePage* message rather than the *enterPage* due to the fact that the book is saved in its default state and thus doesn't need to be initialized on the way in. However, I use *enterPage* here so it is consistent with our other environments.

[13] OpenScript is alone in our environments in the fact that you can't declare and initialize variables on the same line.

[14] Called a *For* loop in ActionScript, JavaScript, and Visual Basic. We'll cover control structures later in the book.

[15] Though you can get a hand cursor when you go to HTML if there is a hyperlink assigned to the object (even if it is just a comment). But we don't want a hand cursor here as that implies that the object can be clicked to make something happen.

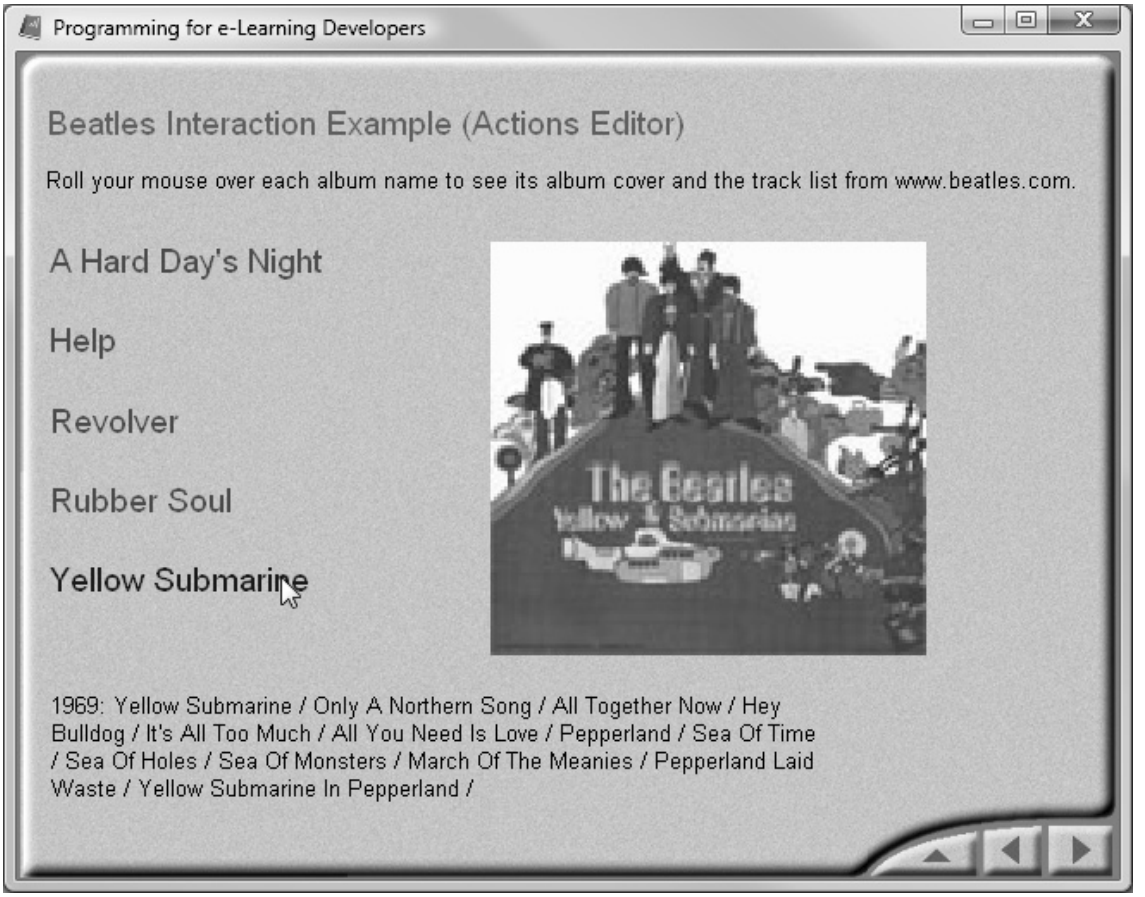

Figure 39. Interactive Example - ToolBook Actions Editor.

Like our “target” example in the last chapter, we will put our actions at the “group” level so we can take advantage of the *target* variable. We initialize and reset by handling the *On load page* event[16] as shown in Figure 40.

[16] Notice that this is an event for the *group*, which is kind of cool. Putting this on the group rather than the page helps us if we copy the group to another page as all its actions will then go with it.

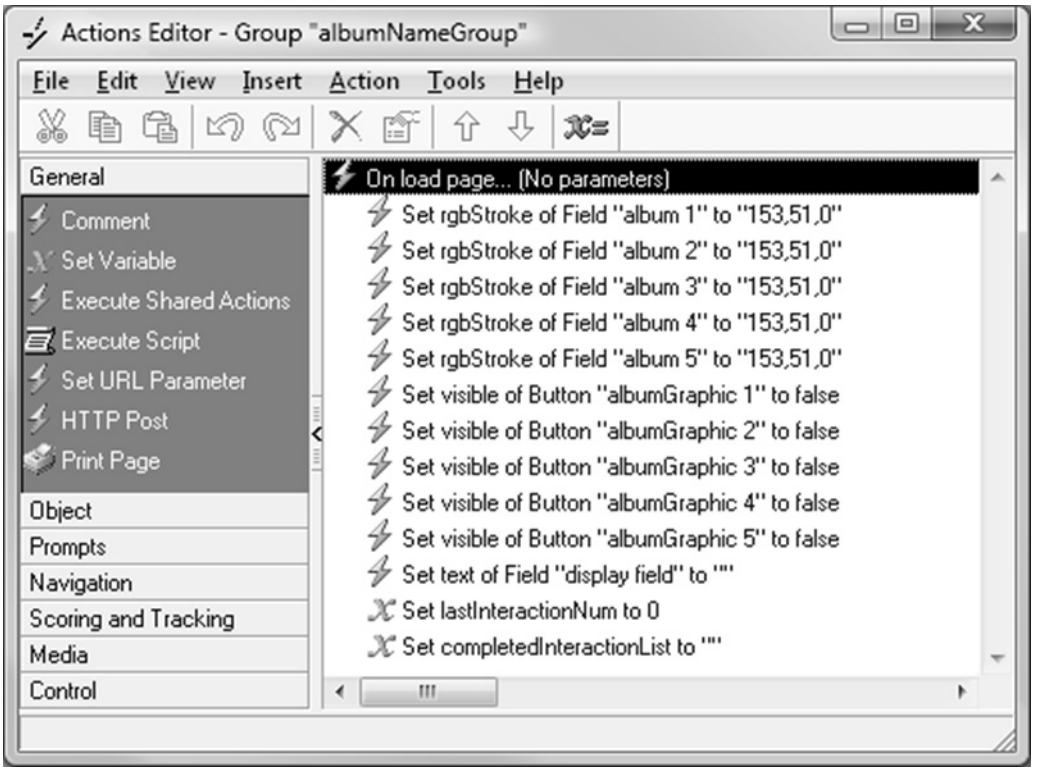

Figure 40. Interactive Example Implementation (On load page): ToolBook Actions Editor.

We set the *rgbStroke* of each of our five album name fields to dark orange[17]. We list each object individually rather than looping through them since we are unable to come up with an object reference dynamically based on the name of the object. Similarly, we hide each of our five album graphics. Finally, we set a *lastInteractionNum* global variable to 0. We use this variable for completion status. The *completedInteractionList* has the same purpose as in our OpenScript example, though we will populate it differently.

We will look at the *On mouse over* code in two parts due to its length. Figure 41 shows the top part of the script.

[17] Notice this is in RGB format while the color in Listing 19 was in HLS format.

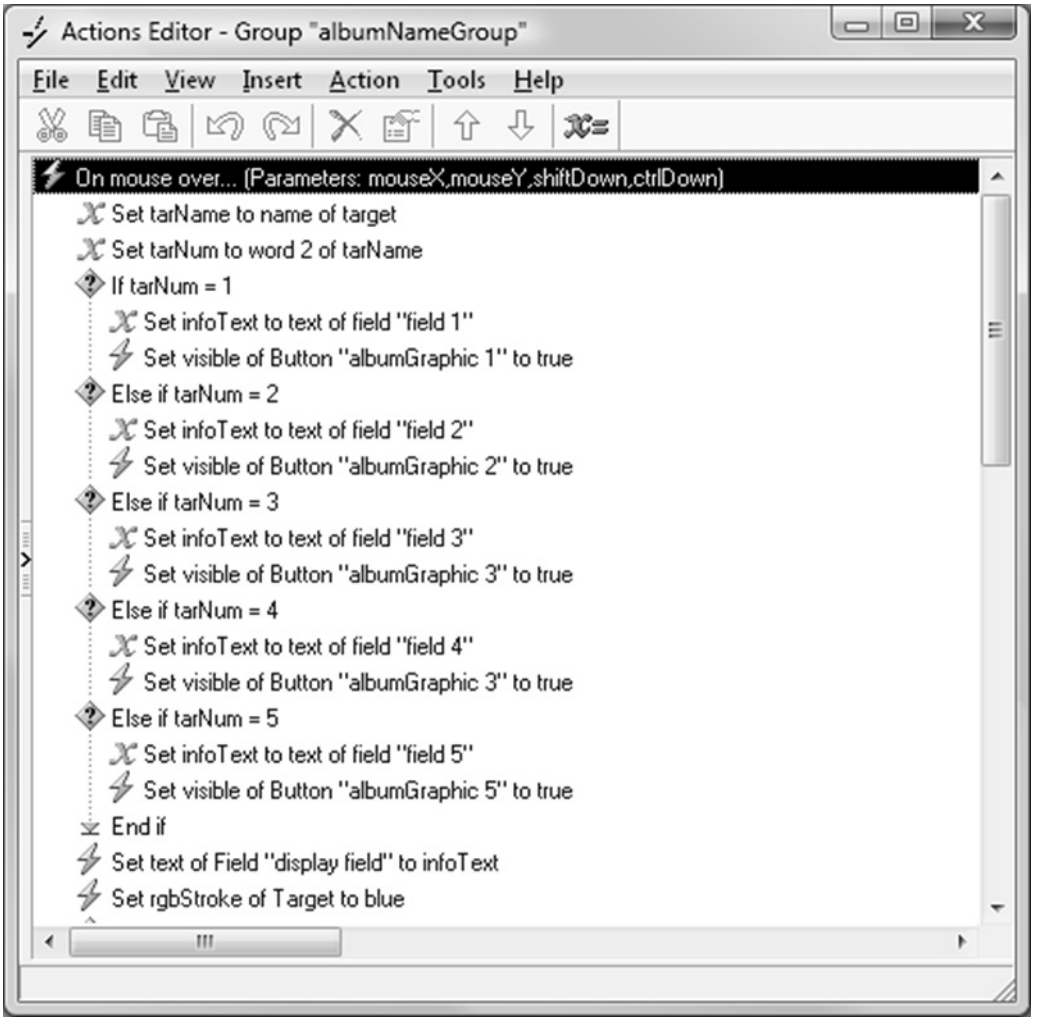

Figure 41. Interactive Example Implementation (On mouse over part 1): ToolBook Actions Editor.

We begin by setting our *tarName* and *tarNum* variables. Notice that the Actions Editor also supports the *word 3 of tarName* syntax, which is why we again separated the base and the number by a space. From there, we use our first *if – else if* construction[18]. Since we cannot do dynamic object referencing, we look at *tarNum* and set a new *infoText* variable with the text of the corresponding field and show its album graphic at the same time. We then set the text of our display field to this *infoText* variable and set the *rgbStroke* of our target field to blue.

[18] In OpenScript, we would use a *conditions* statement for this. In ActionScript and JavaScript, we would use a *switch* statement. In Visual Basic, we would use *Select Case*. We cover all these later in the book.

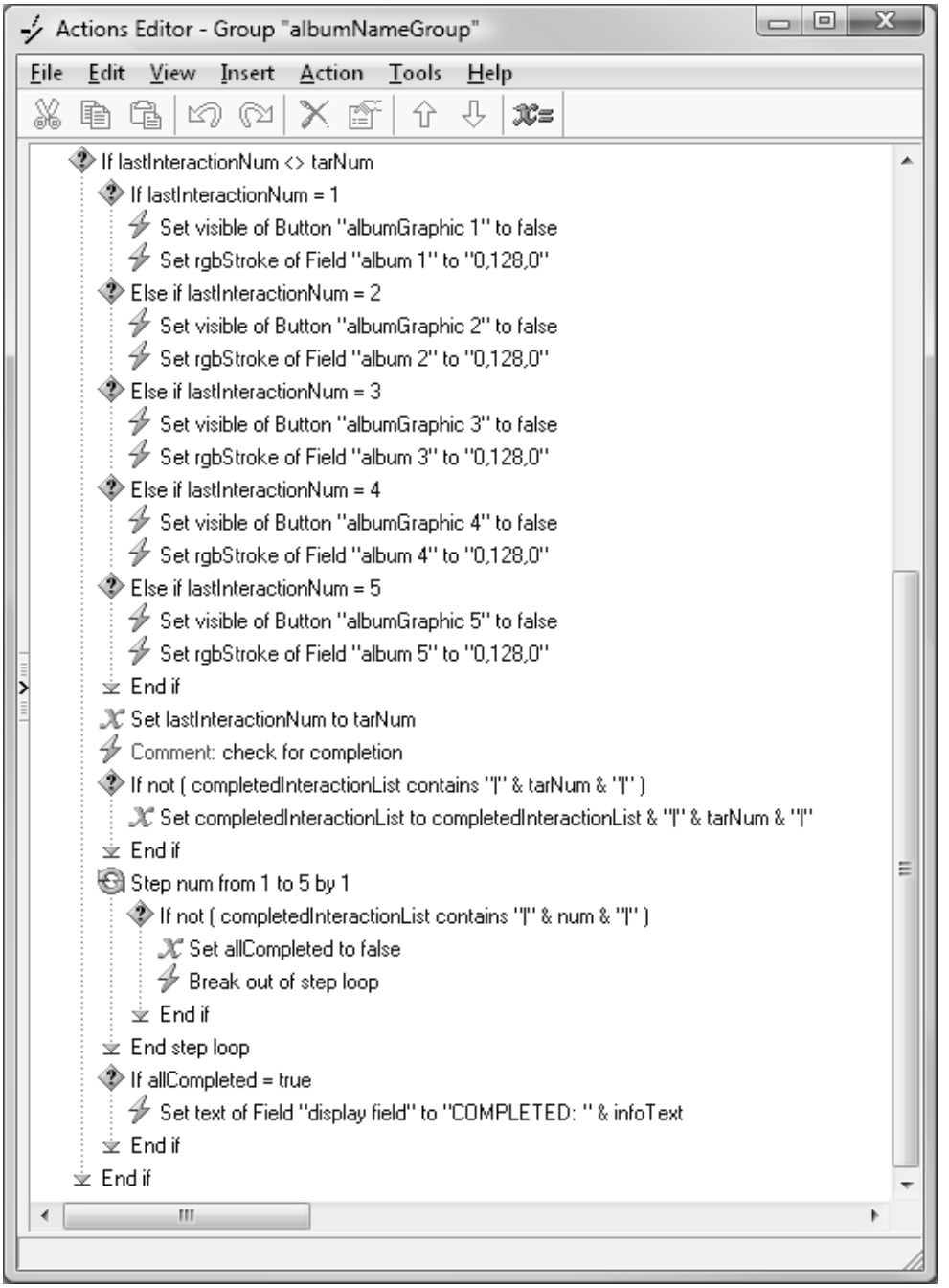

Figure 42. Interactive Example Implementation (On mouse over part 2): ToolBook Actions Editor.

Figure 42 shows the remainder of the logic. We start by checking whether our *lastInteractionNum*

global variable corresponds to the field we are on. If so, we do nothing. If not, we hide the previous graphic and set the *rgbStroke* of that previous field to a dark green. We then set *lastInteractionNum* to match our *tarNum*[19]. We next check for completion. There are a number of approaches to use here. I decided to make a big string of the interaction numbers. But I didn't want to just concatenate the numbers like this: *13254*. The reason is that doing that would run into problems once you got more than 9 interactions as you could not distinguish between 1 and 10. So I put a pipe (|) on either end so that the series would look like this: *|1||3||2|5||4|*. The logic then uses the built-in *contains* function to check for *"|" & tarNum & "|".* We add that sequence to the *completedInteractionList* variable if it is not there and then *Step* through the list interaction number by interaction number. If the sequence is not found, we set our *allCompleted* local variable to *false* and break out of the loop. You can't see it in Figure 42, but we initialized *allCompleted* to *true*. So if we got through our loop without jumping inside the *if* statement, then we can once again add "COMPLETED:" to the front of the text of our display field.

Flash

Let's tackle this same task in Flash. The finished result is shown in Figure 43. We meet all our "requirements" except the swapping of the cursor when the user rolls over the album name. While this is possible in Flash, it is fairly involved[20] and not that important.

[19] This *lastInteractionNum* corresponds to our *lastFieldId* in the OpenScript and other examples. However, you cannot put an object reference into an Actions Editor variable and then use it to set properties of the object. So we just store the interaction number instead.

[20] Basically, you hide the current cursor, show a "movie clip" of the graphic you want for the cursor, and then add some event handlers to make the location of the movie clip match the location of the hidden cursor.

Figure 43. Interactive Example - Flash.

Similarly, the most elegant[21] solution for the album graphics would be to leave them as external files and read them dynamically. We'll show how to do this in a later chapter, but for now we just move them from off the *stage*[22]. Similarly, we take our information fields (*TextField* objects in Flash) off the stage as well. We name these *field_1* through *field_5* as we did in ToolBook, though we use the _ rather than a space to separate the base part and the number since Flash objects cannot have spaces in their names. The album titles are named *album_1* through *album_5.* Our Flash layout is shown in Figure 44.

[21] An elegant solution not only gets the job done but does it efficiently and robustly. So if someone looks at your approach to a task and says that it is "elegant," then that's a real compliment!

[22] The stage is the visible part of the screen, similar to the page in ToolBook and Silverlight. Unique to Flash, however, is the fact that you can move objects off the stage and still see them in the interface. You can move objects off the page in Flash or Silverlight, but then you can't see them. Combined with the fact that you can't set the *visible* property of an object in Flash except in code and you have good incentive to move things off the stage rather than hide them.

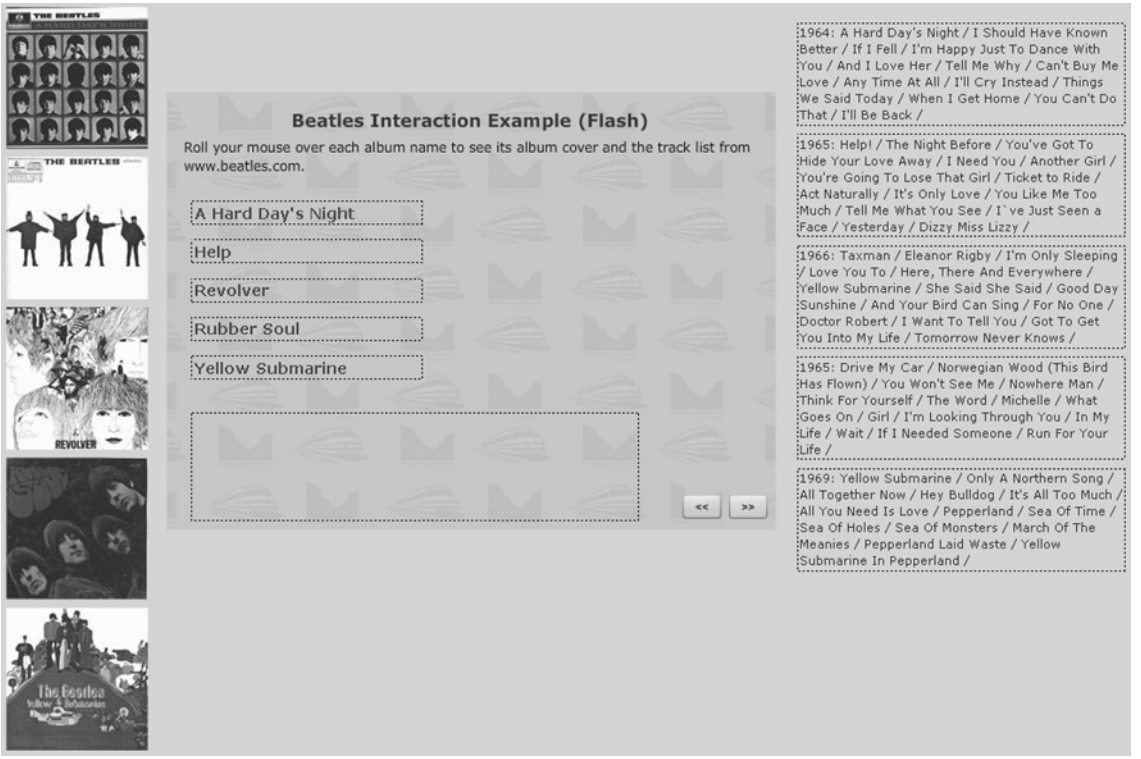

Figure 44. Interactive Example in Flash Showing Objects Moved Off the Stage.

To get the graphics into Flash, we “import” them into the Library and then drag “instances” of them onto the design surface. A slight quirk in Flash, though, is that we cannot refer to these in code unless we change them from *Graphic* symbols to *Movie Clip* symbols. You do this in the Properties window as shown in Figure 45. Once we do this, we name our albums *albumGraphic_1*

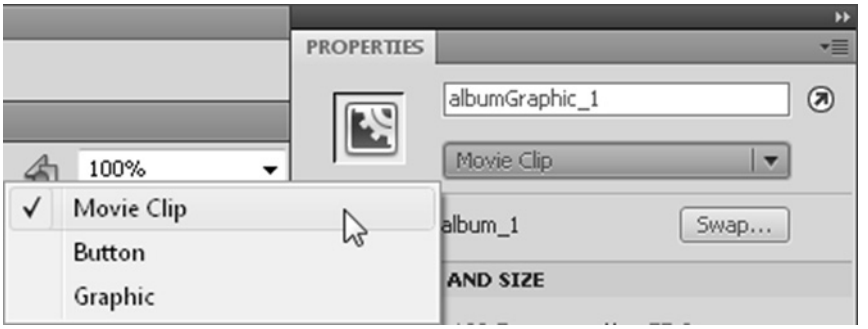

Figure 45. Converting an Instance of a Symbol from a Graphic to a Movie Clip In Order to Refer to It By Name.

through *albumGraphic_5*.[23]

Listing 20 shows our global variables and initialization logic.

```
// "global" variables
var numInteractions:Number = 5;
var lastFieldId:TextField = null;
var completedInteractionList:Array = new Array(numInteractions - 1);
var lastGraphicId:MovieClip;

runInitialLoad3();
stop();

function runInitialLoad3() {
 // set up listeners
 var num:Number;
 var textFieldId:TextField;

 for(num = 1; num <= numInteractions; num ++) {
  textFieldId = this[("album_" + num)];
  textFieldId.textColor = 0x993300; // dark orange
  textFieldId.addEventListener(MouseEvent.MOUSE_OVER,
   implementRollover);
 }
}
```

Listing 20. ActionScript Global Variables and Initialization Code.

We make variables "global" by defining them outside a function block. Notice how we can initialize the variables on the same line where they are declared. This is a nice time-saver. *numInteractions* and *lastFieldId* are used identically to our ToolBook solution. *completedInteractionList* is an *Array*[24] in this case as that ends up being easier to work with. We will basically have a spot for each interaction and will put "true" into the corresponding spot when we complete this interaction. We "dimension" the array as "numInteractions – 1" because we start at 0 and thus want the size to be 4. Here is how the array will look when the user has completed interactions 1, 3, and 5.

[23] We name them slightly different in ToolBook since we don't want the graphic object names to be the same as our field objects. In ToolBook, the graphic resources were not objects on the page so it did not matter if they had the same names as the fields.

[24] I like to think of an array as a table in Word or a spreadsheet in Excel. A "single-dimensional" array is either just one row or just one column. A 2-dimentional array has both rows *and* columns. After that, it gets harder to visualize. "Normal" arrays are accessed by their number (typically starting with 0 except in OpenScript, where the first element is 1). "Associative Arrays" can be accessed via a key that can be numeric or text. We'll see some examples of these later in the book.

true
true
true

Since we will be moving the album graphics into position and then back off the stage, we need another global variable to refer to the "last graphic" we moved into position. We call this variable *lastGraphicId* and declare it as a *Movie Clip* since that's what type of symbols we have (see Figure 45).

In the *runInitialLoad3* function, we define a couple of local variables and then have our first *for* loop. The ActionScript and JavaScript for loops have this format:

for (initial condition; "stopping" condition; increment) {

}

In this case, the initial value of the variable is 1, we keep going as long as *num* is less than or equal to *numInteractions*. After each loop, we add 1 to *num*.[25] Within the loop, we get a reference to our correct album name using the *this[<object name>]* syntax. We put the reference into the *textFieldId* local variable for efficiency and then set its *textColor* property. After that, we add a *listener* like we have seen before. But this time we associate our function (*implementRollover)* with the *MouseEvent.MOUSE_OVER* event. Notice how we call this same function for each of our five album names. Listing 21 shows the code for this function.

```
function implementRollover(eventId:MouseEvent) {
 var tarName:String = eventId.target.name;
 var nameArray:Array = tarName.split("_");
 var tarNum:String = nameArray[1];
 var num:Number;
 var allCompleted:Boolean = true;

 DisplayField.htmlText = this["field_" + tarNum].htmlText;
 if (lastGraphicId != null) {
  lastGraphicId.x = -200;
  lastGraphicId.y = 65;
 }
 lastGraphicId = this["albumGraphic_" + tarNum];
 lastGraphicId.x = 302;
 lastGraphicId.y = 100;
```

[25] *num++* is shorthand for *num = num + 1*. In Visual Basic, we can write the same thing as *num += 1*.

```
  eventId.target.textColor = 0x0000FF; // blue
  if (lastFieldId != null) {
   lastFieldId.textColor = 0x008000; // dark green
  }
  lastFieldId = TextField(eventId.target);
  // check for completion
  completedInteractionList[Number(tarNum) - 1] = "true";
  for(num = 0; num < numInteractions; num++) {
   if (completedInteractionList[num] != "true") {
    allCompleted = false;
    break;
   }
  }
  if (allCompleted == true) {
   DisplayField.htmlText = "COMPLETED: " + DisplayField.htmlText;
  }
}
```

Listing 21. implementRollover Implementation in Flash ActionScript.

As we saw with our OpenScript example, we put our target name (here *eventId.target.name*) into the *tarName* variable. But since we can't put spaces in object names and there is no "word 2 of tarName" syntax, we use the *split* method of strings. It converts a string into an array by pulling out each character (or characters) specified in the parameter ("_" in our case) and put each part into its own element. So the name *album_3* gets split into an array (*nameArray*) that looks like this:

album
3

From there we grab the second element[26] to populate our *tarNum* variable. We then declare our counter (*num*) and a Boolean[27] called *allCompleted*, which we will use to determine if we have finished all the interactions. We set it to true and then will check our *completedInteractionList* array looking for values other than "true." If we find any, we will set *allCompleted* to false and "break" out of our loop.

After the variable declarations, we set the *htmlText* of our display field to the corresponding *htmlText* of the information field. We again use dynamic object referencing with the *this[<object name>]* syntax[28]. We use *htmlText* so that any HTML formatting tags such as <b> and <i> will be copied as well. Next, we check if the *lastGraphicId* global variable has been defined (e.g., if this is the 2nd or later interaction). If

[26] We again start from 0 so the second element is *nameArray[1]*.

[27] A Boolean variable is either true or false. Note that true is not the same as "true" in most languages as the former is a Boolean and the latter is a String. In OpenScript, this type of variable is declared as *logical*.

[28] This is quite helpful but you are on your own to ensure that the object of that name exists AND that it has an *htmlText* property. If not, the Flash movie will only give errors once you start running. It will also do a bunch of weird cycling through all the frames.

so, we move it off the stage by setting its *x* and *y* coordinates[29]. We then set the *lastGraphicId* to the desired graphic and move it onto the stage. Similarly, we set the *textColor* of our album title to blue. If *lastFieldId* is defined, we set its *textColor* to dark green to reflect that the user has already interacted with that title. Either way, we set the *lastFieldId* so that we are all set for the next interaction.

Our last task is the check for completion. We set the corresponding element of our *completedInteractionList* array to "true,"[30] and then use another *for* loop to check each element of this global variable for any values that are not "true." Notice how we initialize to *num = 0* and then have the condition of *num < numInteractions* since the array is zero-based. If we find an element that is not "true," we set *allCompleted* to false and use the *break* command to exit the for loop. Since we initialized *allCompleted* to true, failing to find anything other than "true" will leave that variable as *true*. In that case, we add the word "COMPLETED:" to the front of the information text. Notice the use of the == in the *if* line. In both ActionScript and JavaScript, we need to be careful of the distinction between the logical "equals" (==) and the "assignment operator" (=). In OpenScript, the Actions Editor, and Visual Basic, these are the same. But in ActionScript and JavaScript, they are not. This has bitten me numerous times!

JavaScript

Our JavaScript implementation is fairly similar to what we did in Flash, though it is actually easier. The result is shown in Figure 46. While we have addressed accessibility (the ability for your application to be successfully used by users with disabilities) to some extent by including *alt* tags and so forth, a thorough discussion of that topic is beyond the scope of this book. I would recommend visiting http://www.w3.org/WAI to learn more[31].

[29] In programming, *x* is measured from the left and *y* is measured from the top. So setting *x* to -200 means that we are moving it to the left of the stage.

[30] We set it to "true" rather than *true* since the empty elements are technically not Boolean values.

[31] Thanks to one of my reviewers, Simon Price of the Institute for Learning & Research Technology, for suggesting this link.

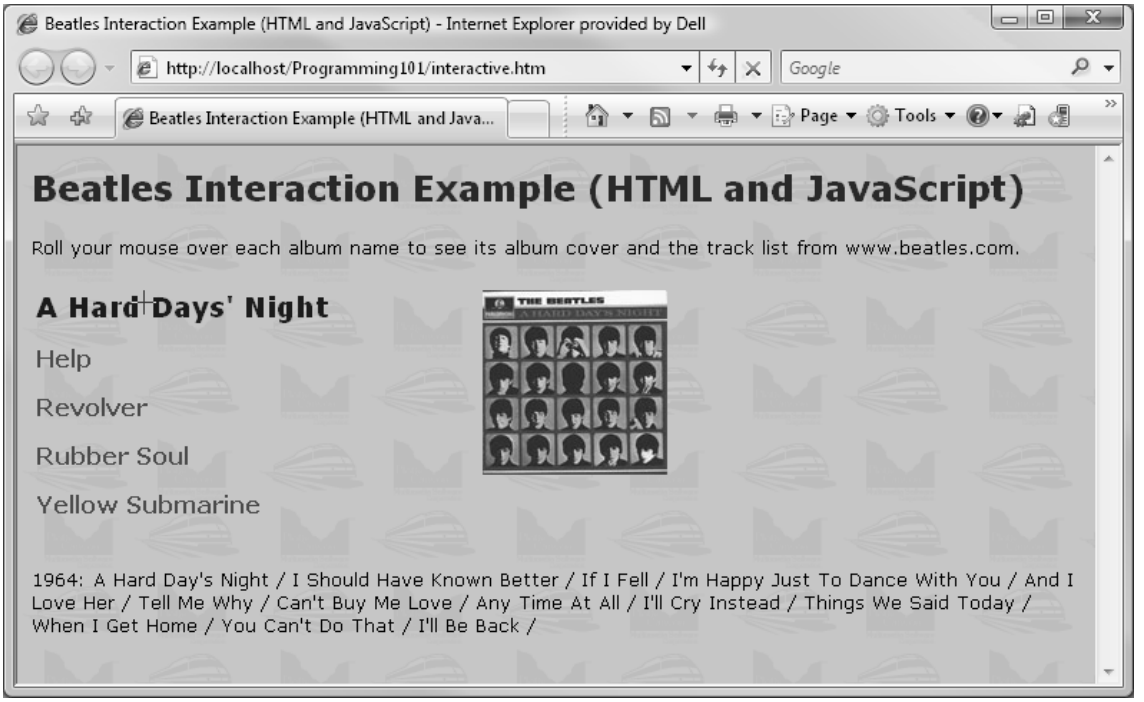

Figure 46. Interactive Example - HTML and JavaScript.

Our code is simplified by the fact that we can use CSS for some of the formatting. The HTML portion of the page is shown in Listing 22. Some key items are shown in bold.

```
<h2>Beatles Interaction Example (HTML and JavaScript)</h2>
<p>Roll your mouse over each album name to see its album cover and the
track list from www.beatles.com.</p>
<table>
 <tr valign="top">
  <td>
   <span id="album_1" class="HotspotReset"
onmouseover="implementRollover(this)">A Hard Days' Night</span>
   <br />
   <br />
   <span id="album_2" class="HotspotReset"
onmouseover="implementRollover(this)">Help</span>
   <br />
   <br />
   <span id="album_3" class="HotspotReset"
onmouseover="implementRollover(this)">Revolver</span>
   <br />
   <br />
```

```
    <span id="album_4" class="HotspotReset"
onmouseover="implementRollover(this)">Rubber Soul</span>
    <br />
    <br />
    <span id="album_5" class="HotspotReset"
onmouseover="implementRollover(this)">Yellow Submarine</span>
   </td>
   <td class="InputButton">
   </td>
   <td>
    <img id="albumImage" src="graphics/album_1.png" alt="A Hard Day's
Night" style="display:none"/>
   </td>
   </tr>
 </table>
 <span id="field_1" style="display:none">1964: A Hard Day's Night / I
Should Have Known Better / If I Fell / I'm Happy Just To Dance With
You / And I Love Her / Tell Me Why / Can't Buy Me Love / Any Time At
All / I'll Cry Instead / Things We Said Today / When I Get Home / You
Can't Do That / I'll Be Back /</span>
 <span id="field_2" style="display:none">1965: Help! / The Night
Before / You've Got To Hide Your Love Away / I Need You / Another Girl
/ You're Going To Lose That Girl / Ticket to Ride / Act Naturally /
It's Only Love / You Like Me Too Much / Tell Me What You See / I`ve
Just Seen a Face / Yesterday / Dizzy Miss Lizzy /</span>
 <span id="field_3" style="display:none">1966: Taxman / Eleanor Rigby
/ I'm Only Sleeping / Love You To / Here, There And Everywhere /
Yellow Submarine / She Said She Said / Good Day Sunshine / And Your
Bird Can Sing / For No One / Doctor Robert / I Want To Tell You / Got
To Get You Into My Life / Tomorrow Never Knows /</span>
 <span id="field_4" style="display:none">1965: Drive My Car /
Norwegian Wood (This Bird Has Flown) / You Won't See Me / Nowhere Man
/ Think For Yourself / The Word / Michelle / What Goes On / Girl / I'm
Looking Through You / In My Life / Wait / If I Needed Someone / Run
For Your Life /</span>
 <span id="field_5" style="display:none">1969: Yellow Submarine / Only
A Northern Song / All Together Now / Hey Bulldog / It's All Too Much /
All You Need Is Love / Pepperland / Sea Of Time / Sea Of Holes / Sea
Of Monsters / March Of The Meanies / Pepperland Laid Waste / Yellow
Submarine In Pepperland /</span>
<br />
<br />
<span id="DisplayField"></span>
```

Listing 22. HTML for Interactive Example.

We use a table to put the album names along the left and have the album graphic show up on the right. Notice how we set the *class* attribute to be “HotspotReset.” Since this CSS class (Listing 23) has the

color and other attributes, we won't need any kind of reset code. The browser takes care of reading the CSS file and setting the properties when the page is reloaded. Note also that we set the *cursor* attribute as part of the class definition as well. As with other implementations, we call the same function (*implementRollover*) for all the hotspots and pass the *this* parameter to get our object reference. After all the album names, we have a "spacer" column and then display an image control in the last column. We have set its *src* attribute for testing but want it to be initially hidden. We do that by setting the *display* style to *none*. We can then use code to set it to *inline* to show it. Below the table are all the information fields. We use the same style setting to keep these hidden. Below that, we have a final span that is our display field.

```
.HotspotReset
{
 color:#993300;
 font-size:125%;
 font-weight:bold;
 cursor:crosshair;
}

.HotspotSelected
{
 color:Blue;
 font-size:150%;
 font-weight:bold;
 cursor:crosshair;
}

.HotspotCompleted
{
 color:#008000;
 font-size:125%;
 font-weight:bold;
 cursor:crosshair;
}
```

Listing 23. CSS Classes for Interactive Example.

The JavaScript to make all this happen is shown in Listing 24. Much of it is similar to the ActionScript implementations of Listing 20 and Listing 21, but without any typing of variables.

```
var numInteractions = 5;
var lastFieldId = null;
var completedInteractionList = new Array(numInteractions - 1);

function implementRollover(targetId) {
    var tarName = targetId.id;
    var nameArray = tarName.split("_");
    var tarNum = nameArray[1];
```

```
    var allCompleted = true;
    var displayFieldId = document.getElementById("DisplayField");
    var infoFieldId = document.getElementById("field_" + tarNum);
    var imgId = document.getElementById("albumImage");

    if (displayFieldId != null && infoFieldId != null) {
        displayFieldId.innerHTML = infoFieldId.innerHTML;
    }
    if (imgId != null) {
        imgId.style.display = "inline"; // visible
        imgId.src = "graphics/album_" + tarNum + ".png";
        imgId.alt = targetId.innerHTML; // tooltip
    }

    targetId.className = "HotspotSelected";

    if (lastFieldId != null) {
        lastFieldId.className = "HotspotCompleted";
    }
    lastFieldId = targetId;

    completedInteractionList[tarNum - 1] = "true";

    for (var num = 0; num < numInteractions; num++) {
        if (completedInteractionList[num] != "true") {
            allCompleted = false;
            break;
        }
    }

    if (allCompleted == true && displayFieldId != null) {
        displayFieldId.innerHTML = "COMPLETED: " +
          displayFieldId.innerHTML;
    }
}
```

Listing 24. Global Variables and implementRollover Implementation in JavaScript

We start once again with our global variables, which we make global[32] simply by defining them outside a function block. *numInteractions*, *lastFieldId*, and *completedInteractionList* have the same role as in the previous examples. Within the function, the biggest difference is that we pass *targetId* as a parameter rather than grabbing a "target." We then get the "name" as the *id* property since that is what is commonly used for HTML controls. We again (like Listing 21) use the *split* command to take our string

[32] It might be worth noting that these variables are only global during the current page and are re-initialized when the user refreshes the page. That's what we want here. But if we wanted them to last for multiple HTML pages, one technique is the have a hidden frame and have the JavaScript variables as part of that frame. We do this with SCORM values in our *Training Studio* (Flash) and *Exam Engine* (Silverlight) products so that we have the data ready to go if the user just closes the browser.

into an *Array* and finally to the *tarNum* variable. We then use our trusty friend, *document.getElementById()* to get references to the display field, the information field for the album we are on, and the image control.

We next check to be sure our object references are not null and set the *innerHTML* of our display field to match that of the information field. Similarly, we show our image control (via the *imgId.style.display = "inline"* line), set its *src* property[33], and set its *alt* tag. We could have done this in our other examples as well but it is most commonly done with HTML content so that screen readers for low-sight individuals have a way to tell what a graphic is. Notice how we set it to the album name.

Rather than setting the color and cursor like we did in other examples, we just set the *className* property to be "HotspotSelected." You can see in Listing 23 that we not only turn the text blue, we also make it bigger by setting the *font-size* to 150%. If our *lastFieldId* is not null, we set it to have a *className* of "HotspotCompleted." This turns it green and sets it back to the original size.

Our last task is to check completion. The logic here is almost identical to Listing 21. We set the correct array value to "true"[34] and then step through the array looking for any values that are not "true." If we find one, we set *allCompleted* to false and break out of the loop. If we have finished all the interactions, we put "COMPLETED:" at the front of the display field. Notice that in this logic we are often checking two things at once: *(allCompleted == true && displayFieldId != null)*. In words, we are checking if *allCompleted* is true AND *displayFieldId* is not null[35].

Silverlight

Four down and one to go! The resulting application, which should be looking familiar by now, is shown in Figure 47. This was one of the easiest implementations yet.

[33] We could have had many images and shown and hidden them like we did in the Actions Editor and Flash example, but it is easier to just change the source to the external image file. This is not possible in the Actions Editor. It is possible in Flash (we'll cover it later) but more challenging than the code we wrote.

[34] We are specifically using the string "true" rather than the logical *true* since the array will be empty initially not truly a Boolean (true or false) value.

[35] *&&* means *AND* and *||* meand *OR* in both JavaScript and ActionScript. This can be confusing to OpenScript. and Visual Basic developers as *&* is a concatenation operator in both languages and *&&* means concatenate and add a space in OpenScript. ActionScript and JavaScript use *+* for both concatenation and addition.

Figure 47. Interactive Example - Silverlight.

The first order of business was to lay out the application. We again take advantage of Silverlight’s “grid” architecture as show in Figure 48 to set up eight rows (one for the title, one for the instructions text, five for the album names, and one for the display field) and two columns (one for the album names and one for the album cover image). This is similar to our HTML table in our JavaScript table but more powerful as we can easily span the title, instructions, and display field across two columns and span the image over the five rows that make up the album names.

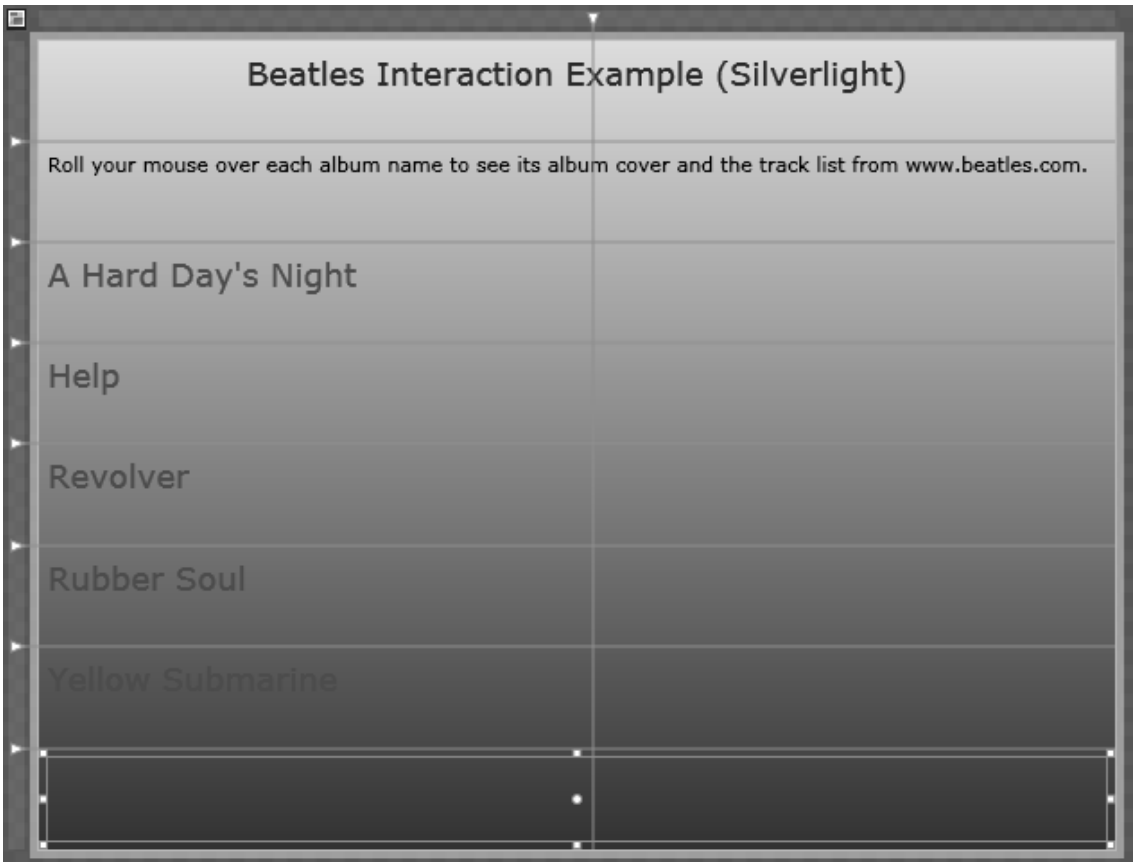

Figure 48. Silverlight Grid in Microsoft Expression Blend.

One of the nice things about Silverlight's XAML is that it is quite convenient to work in the XAML itself. So once I got one *album_1* configured, the quickest way forward was to copy its XAML, paste it four times (for *album_2* to *album_5*) and then edit the things that changed: *Grid.Row*, *Text*, and *x:Name*. The complete XAML is shown in Listing 25. Key settings are shown in bold and explained below.

```
<UserControl x:Class="Silverlight1.Interaction"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Width="600" Height="450"
xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
mc:Ignorable="d">
    <Grid x:Name="LayoutRoot">
        <Grid.Background>
            <LinearGradientBrush EndPoint="0.5,1" StartPoint="0.5,0">
```

```
                <GradientStop Color="#FFDADFE5" Offset="0"/>
                <GradientStop Color="#FF0A3B70" Offset="1"/>
            </LinearGradientBrush>
        </Grid.Background>
        <Grid x:Name="Grid1">
            <Grid.RowDefinitions>
                <RowDefinition/>
                <RowDefinition/>
                <RowDefinition/>
                <RowDefinition/>
                <RowDefinition/>
                <RowDefinition/>
                <RowDefinition/>
                <RowDefinition/>
            </Grid.RowDefinitions>
            <Grid.ColumnDefinitions>
                <ColumnDefinition Width="0.5*"/>
                <ColumnDefinition Width="0.471*"/>
            </Grid.ColumnDefinitions>
         <TextBlock Margin="5,5,5,5" Grid.ColumnSpan="2" Text="Beatles
Interaction Example (Silverlight)" TextWrapping="Wrap" FontSize="18"
Foreground="#FF003366" HorizontalAlignment="Stretch"
TextAlignment="Center"/>
    <TextBlock Margin="5,5,5,5" Grid.ColumnSpan="2" Text="Roll your
mouse over each album name to see its album cover and the track list
from www.beatles.com." TextWrapping="Wrap"
HorizontalAlignment="Stretch" Grid.Row="1" />
    <Image HorizontalAlignment="Left" Width="Auto" Grid.RowSpan="5"
Grid.Row="2" Grid.Column="1" x:Name="albumImage"
Stretch="UniformToFill" VerticalAlignment="Top" Margin="10,10,10,10"/>
    <TextBlock Margin="5,5,5,5" Grid.Row="2" Text="A Hard Day's Night"
TextWrapping="Wrap" Foreground="#FF993300" FontSize="18"
x:Name="album_1" Cursor="Stylus" MouseEnter="implementRollover"/>
    <TextBlock Margin="5,5,5,5" Grid.Row="3" Text="Help"
TextWrapping="Wrap" Foreground="#FF993300" FontSize="18"
x:Name="album_2" Cursor="Stylus" MouseEnter="implementRollover"/>
    <TextBlock Margin="5,5,5,5" Grid.Row="4" Text="Revolver"
TextWrapping="Wrap" Foreground="#FF993300" FontSize="18"
x:Name="album_3" Cursor="Stylus" MouseEnter="implementRollover"/>
    <TextBlock Margin="5,5,5,5" Grid.Row="5" Text="Rubber Soul"
TextWrapping="Wrap" Foreground="#FF993300" FontSize="18"
x:Name="album_4" Cursor="Stylus" MouseEnter="implementRollover"/>
    <TextBlock Margin="5,5,5,5" Grid.Row="6" Text="Yellow Submarine"
TextWrapping="Wrap" Foreground="#FF993300" FontSize="18"
x:Name="album_5" Cursor="Stylus" MouseEnter="implementRollover"/>
    <TextBlock Text="" FontWeight="Normal" x:Name="Display_Field"
VerticalAlignment="Stretch" Height="Auto" Margin="5,5,5,5"
```

```
Grid.ColumnSpan="2" Grid.RowSpan="1" Grid.Row="7" FontSize="10"
Foreground="#FFE3D813" FontFamily="Portable User Interface"
TextWrapping="Wrap"/>
        </Grid>
  <TextBlock Text="1964: A Hard Day's Night / I Should Have Known
Better / If I Fell / I'm Happy Just To Dance With You / And I Love Her
/ Tell Me Why / Can't Buy Me Love / Any Time At All / I'll Cry Instead
/ Things We Said Today / When I Get Home / You Can't Do That / I'll Be
Back /" x:Name="field_1" Visibility="Collapsed"/>
  <TextBlock Text="1965: Help! / The Night Before / You've Got To Hide
Your Love Away / I Need You / Another Girl / You're Going To Lose That
Girl / Ticket to Ride / Act Naturally / It's Only Love / You Like Me
Too Much / Tell Me What You See / I`ve Just Seen a Face / Yesterday /
Dizzy Miss Lizzy /" x:Name="field_2" Visibility="Collapsed"/>
  <TextBlock Text="1966: Taxman / Eleanor Rigby / I'm Only Sleeping /
Love You To / Here, There And Everywhere / Yellow Submarine / She Said
She Said / Good Day Sunshine / And Your Bird Can Sing / For No One /
Doctor Robert / I Want To Tell You / Got To Get You Into My Life /
Tomorrow Never Knows /" x:Name="field_3" Visibility="Collapsed"/>
  <TextBlock Text="1965: Drive My Car / Norwegian Wood (This Bird Has
Flown) / You Won't See Me / Nowhere Man / Think For Yourself / The
Word / Michelle / What Goes On / Girl / I'm Looking Through You / In
My Life / Wait / If I Needed Someone / Run For Your Life /"
x:Name="field_4" Visibility="Collapsed"/>
  <TextBlock Text="1969: Yellow Submarine / Only A Northern Song / All
Together Now / Hey Bulldog / It's All Too Much / All You Need Is Love
/ Pepperland / Sea Of Time / Sea Of Holes / Sea Of Monsters / March Of
The Meanies / Pepperland Laid Waste / Yellow Submarine In Pepperland
/" x:Name="field_5" Visibility="Collapsed"/>
    </Grid>
</UserControl>
```

Listing 25. Interaction.xaml for Interactive Example.

The title, instructions, and display fields span both columns as mentioned above. They do this with the *Grid.ColumnSpan = "2"* setting[36]. The image control has a number of interesting settings. We set its width and height to *Auto*[37] so that it sizes to the space available in its container (here the grid). That space starts in Row 2 and Column 1 and spans five rows: *Grid.RowSpan = "5".* To make the graphic fill up the space available but keep the same dimensions, we set *Stretch="UniformToFill"*. But we don't want the image to go all the way to the edge of the screen, so we set the *Margin* to be 10 pixels on every side: *Margin="10,10,10,10"*. Finally, we want the image to line up with the first album name, giving us *VerticalAlignment="Top"*.

[36] We don't actually need to enter this in the XAML as we can set this in the properties sheet in Blend.

[37] You might notice that there is no height setting. That is because its default value is *Auto.* When a property has its default value, it is not listed in the XAML.

We put each of the album names in the first column and its own row (2 – 6). We set the *Cursor="Stylus"* so that the cursor changes when the user moves her mouse into album name. We tell each of the album name *TextBlock's* to call the *implementRollover* method (shown in Listing 26) in response to the *MouseEnter* event with this XAML: *MouseEnter="implementRollover"*.

We put the information about each album in *TextBlock* controls as well. We keep them hidden with the *Visibility="Collapsed"* settings.

Listing 26 shows the code from the Interaction.xaml.vb file.

```
Private numInteractions As Integer = 5
Private lastFieldId As TextBlock = Nothing
Private completedInteractionList As New List(Of String)

Private Sub implementRollover(ByVal sender As System.Object, _
 ByVal e As System.Windows.Input.MouseEventArgs)

 Dim targetId As TextBlock = CType(sender, TextBlock)
 Dim nameArray As String() = targetId.Name.Split(CChar("_"))
 Dim tarNum As String = nameArray(1)
 Dim allCompleted As Boolean = True
 Dim infoFieldId As TextBlock = _
  CType(Me.FindName(String.Format("field_{0}", tarNum)), TextBlock)

 If infoFieldId IsNot Nothing Then
  Display_Field.Text = infoFieldId.Text
 End If

 ' Build graphic path
 Dim graphicPath As String = String.Format("graphics/album_{0}.png",
tarNum)
 Dim uriId As New Uri(graphicPath, UriKind.Relative)
 Dim graphicId As New BitmapImage(uriId)

 With albumImage
  .Source = graphicId
  .Visibility = Windows.Visibility.Visible
 End With

 targetId.Foreground = New SolidColorBrush(Colors.Blue)

 If lastFieldId IsNot Nothing Then
  lastFieldId.Foreground = New SolidColorBrush(Color.FromArgb(255, 0,
   128, 0)) ' Dark Green
 End If
 lastFieldId = targetId

 If completedInteractionList.Contains(tarNum) = False Then
```

```
  completedInteractionList.Add(tarNum)
 End If

 If completedInteractionList.Count >= numInteractions Then
  Display_Field.Text = "COMPLETED: " & Display_Field.Text
 End If
End Sub
```

Listing 26. implementRollover Code in Visual Basic (Silverlight).

As with most of our other environments, we can share variables by putting them outside a function or sub block. Visual Basic is more precise[38] in that we can set the access level of the variables. *Private* means that methods within the current page can access the variables but no objects outside the page can read or write them. We'll cover access levels later in the book. *numInteractions* and *lastFieldId* have the same meanings as in our earlier examples. *completedInteractionList* has the same purpose but rather than a String or an Array, it is defined as a *List*. This is a type of object called a *Generic*[39] since it can hold different types of objects depending on how we define it. In our case, we'll add the interaction number each time we complete an interaction.

Next up is the *implementRollover* method. The first thing to notice is that our *e* parameter is now of type *MouseEventArgs*. We are not using these arguments in our logic, but we could get things like the mouse position (which we will use later for drag & drop as well as a glossary capability). Figure 49 shows how we can use IntelliSense to get the methods and properties available.

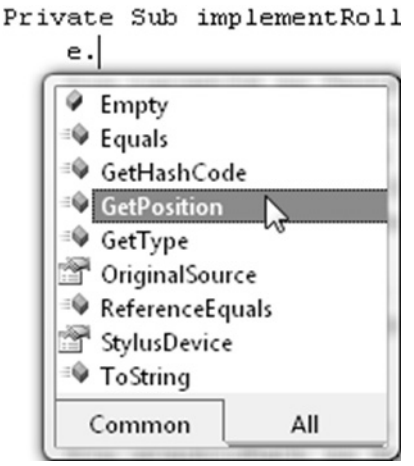

Figure 49. IntelliSense for MouseEventArgs in Silverlight.

As we have seen in previous Silverlight examples, we get our *targetId* from the *sender* parameter. We use the same *Split* technique[40] that we used in ActionScript and JavaScript to separate our name into a base part and the *tarNum*. We use *allCompleted* and *infoFieldId* in the same way as previous examples. Notice the use of the *FindName* method of the page that returns the Silverlight control that matches the name we pass in[41]. Since this could be any type of control, we

[38] ActionScript can also be precise in setting variables and functions to private, public, etc., but only within ActionScript classes.

[39] Generics always have the "Of" part of the definition. The cool part is that we can have a list "of" whatever type of object we want. Here it is just a list of Strings. But it could be a list of bitmaps, question objects, integers, etc. Visual Studio will then tell us if we try to put a different type of object into the list or if we take something out a list and try to use it incorrectly. This helps to reduce bugs quite a bit.

[40] The Visual Basic *Split* method takes a *Char* parameter rather than a *String*. That is why we have to use *CChar("_")* as the parameter. Although these "strong types" can be a bit of a pain sometimes, believe me that it saves time and helps you write better code.

[41] This is a very nice addition with Silverlight. There is an equivalent to *FindName* in ASP.NET but not in .NET Windows Forms. I spent quite a bit of time in my VBTrain.Net book explaining how to use techniques like "Reflection" to do dynamic object referencing. Luckily, the *FindName* method takes care of it for us in Silverlight.

use our handy *CType()* to tell the compiler that we know it is a *TextBlock*. If we could not find a control of that name, *infoFieldId* will be *Nothing*. So we check that *infoFieldId IsNot Nothing* before setting the text of the display field.

Like our HTML and JavaScript example, we will load the graphics dynamically. But rather than putting them into a subdirectory of our web site[42], we can actually build the graphics right into our DLL. To do that, we add them to our project and set their *Build Action* to *Resource*[43] as shown Figure 50. To get our hands on the graphic, we first construct the relative *graphicPath* from the *tarNum*. If we had chosen a *Build Action* of *Content* instead, the only difference would be that we would use a leading / in the *graphicPath*. From there we create a new *Uri* [44] using the *graphicPath* and a parameter designating that this is a relative path. From there, we create a new *BitmapImage* from the Uri. Finally, we set the *Source* of the image to this *graphicId* and show the image. Note the use of the *With* syntax to set multiple properties or call multiple methods. This avoids having to put "*albumImage.*" over and over.

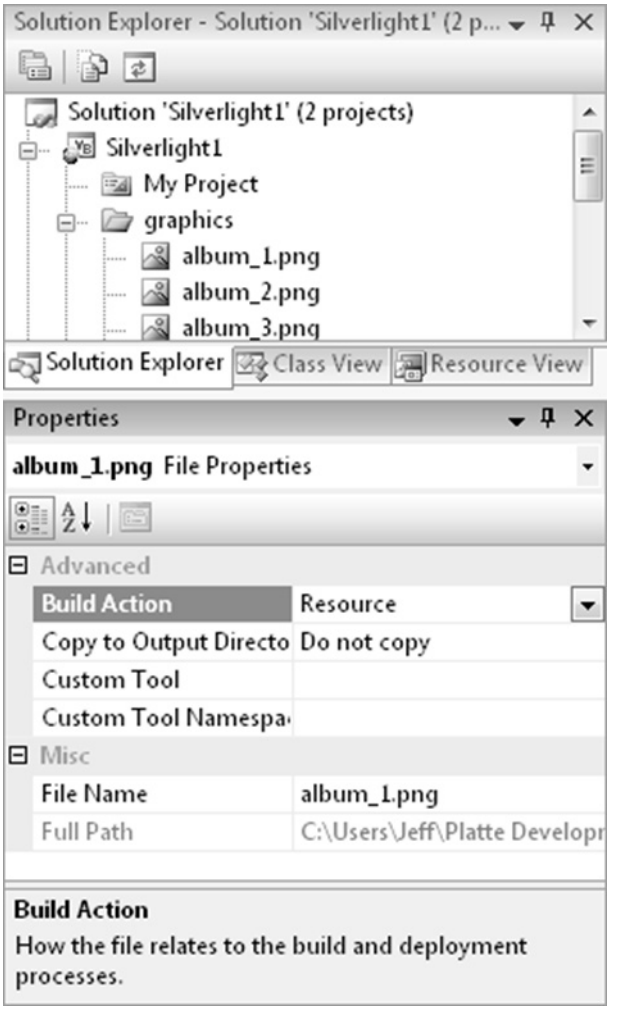

Figure 50. Setting Graphics to Resources in Visual Studio.

To set the color of our album name, we return to our *SolidColorBrush* that we encountered earlier. After checking to make sure our *lastFieldId* exists (i.e., we are on at least our 2nd interaction), we set its color to dark green using the *Color.FromArgb()* method. Note the "A" in there, which gives us an alpha channel, or transparency, component to the color.

[42] You can load graphics via an *absolute* URL as well in both HTML and Silverlight.

[43] A *Build Action* of *Resource* puts the items right inside the DLL. A *Build Action* of *Content* puts the files into the *.xap* file instead. We use *Content* in *Exam Engine* so that we can build the .xap file programmatically without having to recompile the DLL.

[44] Uniform Resource Identifier used to name or identify a resource on the Internet. Most of us would call this a URL, but technically a URL also has a location. See http://en.wikipedia.org/wiki/Uniform_Resource_Identifier.

Our completion logic is most similar to our OpenScript implementation back in Listing 18. We use the nice *Contains* method of our *List* to see if it already has our *tarNum*. If not, we add it to the list. Since we can then be sure that each interaction number only is listed once, we can just check the *Count* property to see if we have completed all the interactions. If so, we add “COMPLETED:” to the front of the display field text.