

APPENDIX B: LOCALIZATION

The Internet has brought people from different countries and cultures together like nothing else in the past century. As content on the Web becomes more and more universally viewed around the world, chances are that a game you develop may need to be localized to other languages. In this appendix, we will walk through an example of how support for many other languages can be added relatively painlessly to an existing game.

Key Points to Remember

When you're working on a game that you think will need to be localized, there are a number of UI (user interface) considerations to remember that will prevent headaches later.

Use Fonts That Support Larger Character Sets

Although all the main fonts that are common across operating systems (like Arial, Verdana, Times, and so on) support very complete character sets, many individual-created font sets only support the basic English alphabet. This becomes problematic when moving to just about any other language because letters with accents or entirely new characters are not included. When selecting a font, do your best to find out through a site like www.fonts.com if the typeface supports the character sets you'll need. If you're localizing for the common Latin-based languages (English, French, Italian, Spanish, and German), you'll want the font to support the sets known as Latin I and Latin Extended A. Most fonts made by professional foundries should at least include these sets.

Always Leave Ample Room for Text Fields

Whether you are working with an artist or designing the game yourself, there's a natural inclination to size and kern the text in an interface, so that it looks proportional and doesn't leave a bunch of excess empty space. However, words or phrases in languages other than English are very likely to be longer and, therefore, require more space. You want to allow as much room as possible for this text to avoid getting cropped or becoming unreadable because of its small point size.

A good rule of thumb, especially if you know the languages you are localizing to, is to use a translator Web site like Google Translate or Babelfish to find the longest possible version of a word or phrase and size to fit that. This way, you're prepared for the worst-case scenario.

Use Scrollbars for Long Sections of Text

In any sections that house a lot of copy, like a rules page, it's best to go ahead and plan for a scrollbar, even if the English copy doesn't require it. If you use the built-in CS4 UI component scrollbar, then it will auto-hide itself, when it is unnecessary. This allows you to not worry about the copy getting too long to have to frame it graphically.

Use Icons Instead of Text wherever Possible

If you have a button in your interface that sends a link to the game to a friend, you could probably easily get away with giving the button an iconic symbol (such as an envelope) instead of typing out "Email a Friend." Any places, where you can reduce your reliance on text, altogether, the less concessions you'll have to make in the look and feel of your interface.

As an addition to this technique, the use of a tool tip (the small box that appears when you roll over a button) is also a good idea. You generally don't have to worry about the length of the text in a well-designed tool tip, so it won't matter if the tip's text is considerably longer in another language.

Localization in Flash CS5

Once you have made design considerations in your game to incorporate localized text, Flash provides a very convenient and powerful way to implement other languages. It provides this functionality through two components: the Strings panel in the IDE and the Locale class in ActionScript. To demonstrate how these two pieces are implemented, we will localize the MixUp game we created in Chapter 13. The files for this example are in the Appendix B folder, specifically the MixUp.fla file.

The Strings Panel

Under the Window > Other Panels menu in Flash CS5, you'll find the Strings panel. This tool allows you to assign any dynamic text field an ID name, which will associate localized text with it. Figure B.1 shows how the panel looks in the IDE.

To begin the process of setting up other languages, click the Settings button in the Strings panel. It will bring up a panel shown in Fig. B.2.

On the left-hand side of the panel are a number of common languages that are built into Flash. You can select one and click the Add button to add it to your document's language list. Once you have selected the languages you want to support (in our



Figure B.1 The Strings panel is used to assign IDs to dynamic text fields.

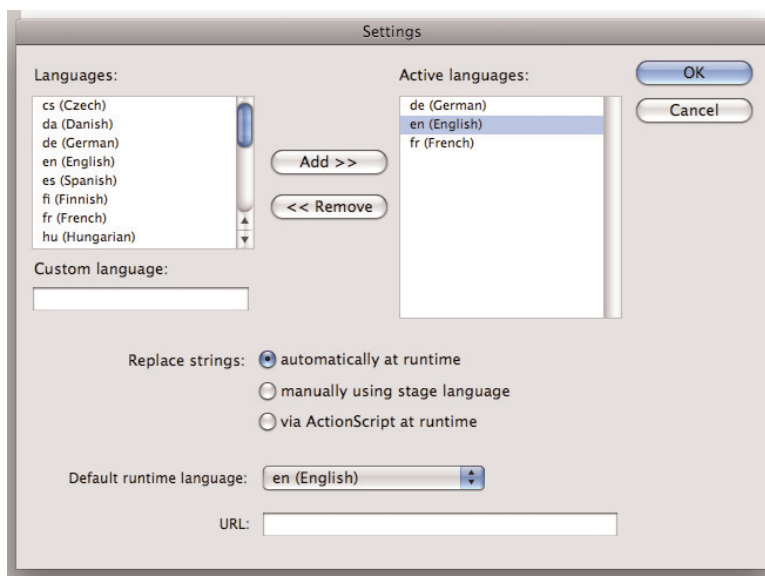


Figure B.2 The Strings Settings panel allows you to configure the languages you will support in your game.

example, just English, French, and German), select the Default runtime language from the combo box at the bottom of the panel. For our purposes, we'll start out with English. Finally, Flash needs to know how we want the localization data to be used in our game. There are three options available:

- *Replace strings automatically at runtime:* This option will store all the localized text in separate XML files for each language and automatically load the correct one based on the language detected by the Flash player. It will also automatically replace any strings that you have tagged for localization.
- *Replace strings manually using Stage language:* If you select this option, then Flash will export the game with one language

embedded in the file and will not utilize the external XML files. If you wanted to support a number of different languages, you'd need to switch the language and publish to separate SWF files. For content that, for whatever reason, will not have access to any of the external files, this is a good option for keeping the files completely self-contained.

- *Replace via ActionScript at runtime:* The best option, and the one we'll be using in our example, will turn over language selection and replacement to us to control through code.

Once the languages are defined, the Strings panel will populate with fields for each locale. You are now ready to begin assigning strings to your text fields. In the case of our MixUp game, I have already prepped the FLA by changing all of the text fields in it from static to dynamic. This is because when static text is compiled, Flash simply reduces it to shapes. In order for it to stay text at runtime, it must be defined as dynamic text. Once you have changed a text field to be dynamic, you want to make sure to embed the font you're using. In the Text Properties panel, select Character Embedding. It will bring up the panel seen as Fig. B.3.

For the languages we're using for MixUp, we need to embed the Latin I and Latin Extended A sets. We'll go ahead and also include the Latin Extended B and Latin Extended Additional sets because German can occasionally contain a rogue character from these sets as well. Now, we're ready to begin assigning text. To tag a text field, simply select it and give a unique name in the ID field, as shown in Fig. B.4.

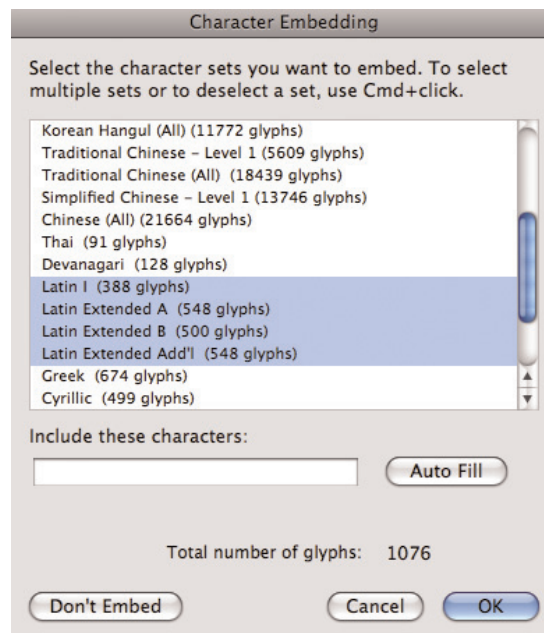


Figure B.3 The Character Embedding panel allows you to select the character sets you want to include.



Figure B.4 Each localized text field has an associated ID to uniquely identify it.

No matter what you enter as an ID for a text field, Flash will convert it to uppercase and prefix it with “IDS_”. This conforms to the XML localization standard that Flash follows. In the String field, you can now enter whatever text you want to use for that field, and it will be bound through its ID. It’s also worth noting that two or more TextFields can share the same ID; if you are using the same string in two places, reuse IDs to save space in the XML file and make translation easier.

After you have repeated this process in English for all the text fields in a game, you have two different options for filling in the other languages. One is to merely type or paste text into the proper column for each respective language and ID. This is relatively easy to do if you are the person responsible for the translations. When you export the SWF, Flash will generate an XML file for each language with all of the text. However, if you are looking to someone else to perform the translations for you, you have another option. Once all the English tags are created, export the SWF. This will generate the XML files, with blanks left in the other languages. The English file will look like this:

```
<xliff version="1.0" xml:lang="en"
  <file datatype="plaintext" original="MixUp.fla" source-
    language="EN">
```

```

<header></header>
<body>
  <trans-unit id="001" resname="IDS_CLOSEBUTTON">
    <source>Close</source>
  </trans-unit>
  <trans-unit id="002" resname="IDS_FINALTIME">
    <source>Final Time:</source>
  </trans-unit>
  <trans-unit id="003" resname="IDS_HOWTOPLAY">
    <source>How To Play</source>
  </trans-unit>
</body>
</file>
</xliff>

```

Note how each ID gets its own “trans-unit” XML node. The lines in bold are the individual strings used for each ID. You can hand over these XML files to your translator and have them fill in the other languages from the English file. After these new files are complete, you can import them from the Strings panel in Flash. Select the Import XML button, and Flash will prompt you for the language you want to import, filling out the entire column. Pretty snazzy, huh? Once all the languages are set up properly, Flash also gives you a handy way of previewing the other languages in context. Simply change the Stage Language in the combo box on the Strings panel. Any localized TextFields that you are currently viewing will update to the selected language. This allows you to check for any overflow or spacing issues. Be sure to switch the language back to English (or the desired default language), when you’re done checking.

At this point, we’ve discussed how to create all of the localizations and generate the required XML for them. Now, it’s time to add some ActionScript that will make the translations active in our SWF. Because we’ll need to wait for the language XML to load before we display any text in the game, I’ve added a blank frame to the beginning of the MixUp.fla main timeline and given it a label of “init.” Correspondingly, there are some additions to the MixUp document class; the additions are in bold below.

```

static public const FRAME_INIT:String = "init";

public function MixUp() {
    enumerateFrameLabels();
    addEventListener(FRAME_INIT, loadLanguage, false, 0, true);
    addEventListener(FRAME_TITLE, setupTitle, false, 0, true);
    addEventListener(FRAME_GAME, setupGame, false, 0, true);
    addEventListener(FRAME_RESULTS, setupResults, false, 0, true);
    createImagePool();
}

```

```
protected function loadLanguage(e:Event):void {
    stop();
    var language:String = Capabilities.language;
    Locale.loadLanguageXML(language, languageLoaded);
}

protected function languageLoaded(success:Boolean):void {
    Locale.autoReplace = success;
    gotoAndStop(FRAME_TITLE);
}
```

When the “init” frame is hit at the beginning of the game, it polls the Capabilities object for the active language of the player. You could also easily force this to a specific language or tie it to a series of buttons allowing the user to, manually, select the language. The Locale class is then used to load the language XML. Once the load is complete, regardless of success, the callback function of *languageLoaded* is called. If the language file is found, we want the Locale class to automatically replace any tagged text on the screen with text from the XML. If the file was not found for some reason, we want to leave the text in its default state (in this case, English).

We are now done with the localization. If you were to change the XML loaded to “de” or “fr” in the MixUp class file, you would see all the text in the game reflect those languages. However, one side effect of changing all of our text fields to dynamic is that anything underneath those fields won’t receive mouse events. This is problematic because on multiple screens, a text field sits on top of a button. By default, TextField objects have their *mouseEnabled* property set to true. Because there is no way to change this default behavior, we must manually set this property to false for all TextField objects. Don’t panic, though; I have created a generic method to do this for us with only a single line of code. Alongside the other classes in the folder is a new class called *TextFieldUtil.as*. In it, there is a static method called *disableTextFields*.

```
public class TextFieldUtil {

    static public function disableTextFields(displayObject:
        DisplayObjectContainer):void {
        for (var i:int = 0; i < displayObject.numChildren;
            i++) {
            if (displayObject.getChildAt(i) is
                TextField) {
                var tf:TextField = displayObject.
                    getChildAt(i) as TextField;
                if (tf.type == TextFieldType.
                    DYNAMIC) {
                    tf.mouseEnabled = false;
                }
            }
        }
    }
}
```

```
        tf.tabEnabled = false;
    }
}
}
```

This function iterates through the display list of a given container, finds all dynamic TextField instances and disables mouse and tab input. This way, our classes that use localized text fields can merely make a call to:

```
TextFieldUtil.disableTextFields(this);
```

Now MixUp is fully set up to support English, French, and German audiences. It should be noted that, in real time, it took me about an hour to tag all the text fields *and* look up the translations for just those two languages. In development terms, that is a minimal investment! There are a number of additional capabilities of the Locale class, such as getting the string for a specific ID or assigning an ID to a TextField at runtime. The former is particularly useful if you need to assign changing text to a field during gameplay, which we don't do in MixUp. To further explore the Locale class, consult the Flash CS5 documentation.