MAKING ON-LINE SCIENCE COURSE MATERIALS EASILY TRANSLATABLE AND ACCESSIBLE WORLDWIDE: TECHNICAL CONCERNS

Christopher V. Malley, *PixelZoom, Inc., Boulder CO, USA*
Jonathan Olson, *Department of Physics, University of Colorado, Boulder, CO 80309, USA*

Introduction

This paper is a supplement to [1], and provides technical details on the internationalization and localization of PhET simulations (sims). PhET sims are implemented in Java or Flash, and supported on Windows, Macintosh and Linux.

Locale Specification

In computing, a *locale* is a set of parameters that defines the user's language, country, and any special variant preferences that the user wants to see in their user interface [2]. For PhET simulations, a locale is defined by a *language code* and an optional *country code*.

There are several different standard conventions for specifying a locale, and the first decision is which convention to use. PhET specifies a locale using codes defined by the International Standards Organization (ISO). ISO 639-1 defines language codes [3], while ISO 3166-1 defines country codes [4]. While there are other standards, these are the most commonly-used definitions. An example of a language code is pt_BR; the language is Portuguese (pt) and the country is Brazil (BR), separated by an underscore. The country portion of the locale is optional for languages where localization does not differ by country.

PhET's early attempts to specify locale used only a language code. Feedback from translators quickly told us that language code alone is inadequate; regional differences often require different translations for the same language. For example, Chinese (language=zh) requires different translation for China (country=CN for Simplified Chinese) and Taiwan (country=TW for Traditional Chinese). So we need to be able to specify zh_CN and zh_TW as separate locales.

Internationalization of Simulation Software

Internationalization is the process of designing software so that it can be adapted to various locales without engineering changes [5]. Internationalization is often abbreviated as *i18n,* because there are 18 letters between 'i' and 'n'. I18n support is part of PhET's common framework, code that is shared by all simulations. Additional i18n support is also provided by Java and Flash.

The first task of i18n is to read and interpret the locale. For Java, the most common method of reading a locale is via standard system properties that identify the locale to the Java Virtual Machine (JVM), which enables Java's built-in i18n support. For Java sims delivered via Java Web Start (JWS), we do not have permission to set the locale for the Java Virtual Machine, and are therefore unable to use much of Java's i18n support. (We are investigating signing JAR files with digital certificates. Doing so would allow us to set the locale for the JVM and take advantage of Java's i18n support.) Instead of using standard Java system properties, we use PhET-specific properties to specify the locale, and provide our own implementation to read and interpret the locale.

Flash handling of locales is similar to Java. Flash's built-in locale support only identifies language, and is not capable of identifying country code. So we have implemented our own mechanism (using FlashVars) to specify and interpret the locale. As with Java, this prevents us from taking advantage of Flash's i18n support.

Once a locale has been identified, a simulation must decide what needs to be adapted to fit that locale. Many things can vary by locale, including language (e.g., alphabets, fonts, writing direction), writing conventions (e.g., date formats, number formats, equations) and cultural differences (e.g., symbol meanings, significance of colors). Java and Flash provide direct support for some of these things, while other things must be addressed by the developer. PhET simulations currently address variables related to language; other variables are not generally supported, or are supported on a sim-by-sim basis. We will look at how PhET simulations support i18n of strings (collections of characters from a specific alphabet), fonts, and writing direction.

Strings are internationalized by replacing literal strings with symbolic key values. Given a locale and a key value, a lookup is performed at runtime, and a locale-specific string is returned. If no locale-specific string is found, the default (English) string is used. For example, code that is not internationalized might look like this in Java (Flash code is similar):

```
String playButtonLabel = "Play"; // a literal string
```

Internationalized code would look like this:

```
String playButtonLabel = TranslatedStrings.lookup( locale,
     "button.play" ); // a lookup
```

Translated strings are stored in locale-specific *string files*. For Java simulations, string files are Java properties files; for Flash simulations, they are XML files. (The file types for Java and Flash sims differ for technical reasons. Java's i18n support typically uses properties files, while Flash has better support for reading XML.). String files contain key/value pairs, with the values using characters from the Unicode standard [6]. In Java properties files, all characters that are not in the Basic Multilingual Plane are in an escaped form, while the characters in Flash string files are directly encoded in UTF-8.

The Java string file entry for the example above would look like this:

```
button.play=Play
```

And the Flash entry would look like this:

```
<string key="button.play" value="Play"/>
```

Each simulation has its own set of string files, one file for each supported locale. Strings that are common to many sims are stored in a special common-strings file, so that they only need to be translated once.

Once we have a translated string, it must be rendered using an appropriate font, and in the proper writing direction (left-to-right, right-to-left, or mixed direction). All PhET sims rely on the user's computer to provide fonts. To select an appropriate fonts in Java sims, PhET maintains a table of preferred fonts. If a preferred font is specified for a locale, and that font is found on the user's computer, then that font is used to render strings. If no preferred font is found, then the default font on the user's computer is used. The default font may or may not be appropriate (e.g., fonts that do not contain Japanese characters are inappropriate for Japanese translations). If an inappropriate font is used, strings may be rendered improperly. In both Java and Flash simulations, for example, characters that cannot be rendered properly appear as rectangles. PhET does not attempt to distribute fonts with simulations, and relies on translators and users to report font problems and assist in identifying preferred fonts.

For Java simulations, the writing direction of strings is handled automatically by Java's text renderer. Based on the specific Unicode characters, the renderer assembles the characters in left-to-right, right-to-left or mixed direction. Flash lacks the native ability to handle writing direction, and instead detects and uses the default writing direction of the user's computer (e.g., a Windows computer set to English will display Arabic characters in an incorrect left-to-right direction). This unfortunately means that for right-to-left strings to display properly for Flash simulations, the user's computer must have right-to-left language support downloaded and have the system set to a locale that has right-to-left text.

Some strings contain *placeholders* that cannot be filled in with text until runtime. These strings present additional writing-direction issues that must be addressed. For example, consider the string "__ is starting up." where the blank will be filled in with the simulation's title. To give the translator control over the order of the words in these types of strings, PhET uses Java's MessageFormat syntax to specify parameterized strings in both Java and Flash. In English, the above string would be "{0} is starting up." Placeholders are specified as {0}, {1}, {2}, etc. and are filled in by the program at runtime. A pitfall of this approach is that it is sometimes difficult for the translator to determine the semantics of the placeholders.

Localization and the PhET Translation Utility

Localization is the process of adapting internationalized software for a specific locale by adding locale-specific user interface components and translating text [5]. PhET localization is limited to translation of strings; we do not provide general support for adding locale-specific components to simulation interfaces

Translation of strings is supported by the PhET Translation Utility. (This utility and instructions for using it are available at http://phet.colorado.edu/contribute/translation-utility.php). Translation Utility provides a translator (a person creating a locale-specific translation) with a GUI (Graphical User Interface) for creating PhET string files. Translators choose a simulation and their locale from a list of locales, as shown in Figure 1. The GUI then displays two columns, as shown in Figure 2: a left column shows the English strings, and a right column allows translators to enter the corresponding strings for their locale. A translation can be tested at any time by pressing a "Test" button, which will run the simulation in the specified locale, with the translator's strings. The translator should be checking that their translation is appropriate, is rendered properly, and does not create user-interface layout problems. Layout problems can occur when a translated string is significantly longer or shorter than the original English string, as shown in Figure 3. When the translation is completed, pressing a "Submit" button creates a string file, which can then be emailed to PhET.

Submissions are reviewed by PhET, and then published to the PhET website. When PhET reviews a submitted translation, we check to see that it runs, is complete, and the layout is acceptable. However, we are not qualified to check the semantic accuracy of the translation. We depend on our user population to notify us of problems – similar to Wikipedia. So far, we have received only serious attempts at translating simulations.

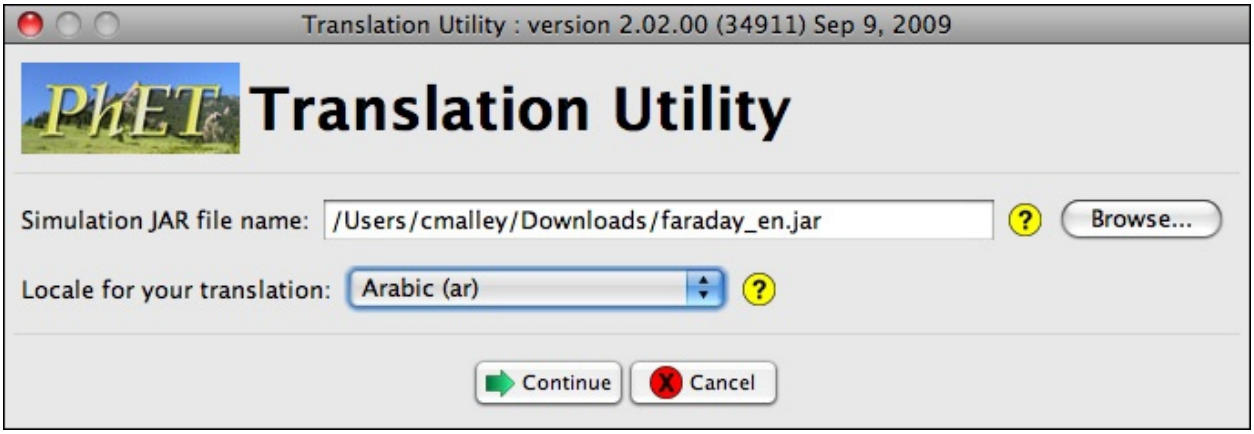

Figure 1: Translation Utility, user interface for selecting simulation and locale

Figure 2: Translation Utility, user-interface for translating string

Online and Offline Delivery

In general, all delivery of PhET sims is locale-specific. When the user selects a sim to run, they also select the desired locale. The sim therefore ignores the locale of the user's computer, and runs in the locale specified by the user. This is different from typical software models, where the application runs in the locale of the user's computer. This difference presents many challenges, and precludes the use of many built-in i18n features of Java and Flash.

PhET bundles all individually-downloadable simulations (Java and Flash) as locale-specific JAR files. Each JAR file contains string files for all locales supported by a simulation. PhET-specific command-line arguments are used to specify the locale at runtime.

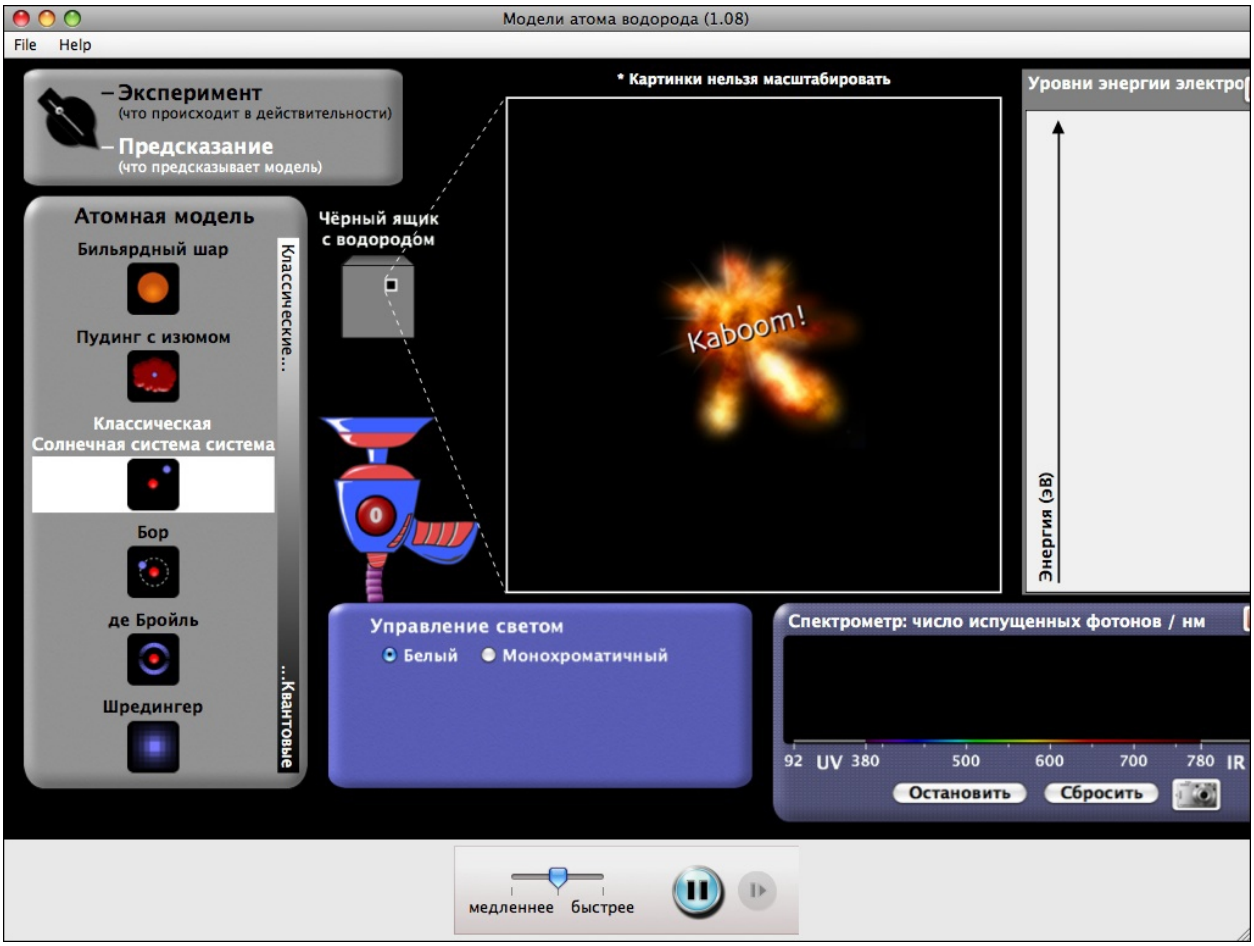

Figure 3: A simulation whose layout has been negatively affected by long translated strings. In this Russian translation, a long string in the left-most control panel results in the right-most parts of the user-interface being pushed off screen.

For *online delivery,* Java simulations are run via locale-specific JNLP files, executed by Java Web Start with translated strings bundled in the JAR file. The JNLP file sets the command line arguments that specify the locale. Flash simulations are run via locale-specific HTML files, with the XML of translated strings embedded in the HTML and passed to the simulation via FlashVars.

For *offline delivery*, Java and Flash simulations are delivered as locale-specific JAR files. Each JAR file contains code that sets the proper locale at startup.

Open Issues

As mentioned previously, there are some aspects of i18n that are not addressed by PhET simulations. Number formats, for example, are not handled properly. Some of these issues would be addressed by using Java's standard method of setting the locale for the Java Virtual Machine (JVM). If the JVM knows the locale, then things like number formats will be automatically handled by Java. Passing this information to the JVM requires the JNLP files

request permission to set Java System properties, and this in turn requires online users to accept a digital certificate. PhET originally felt that acceptance of digital certificates could pose a barrier to simulation use, but certificates are becoming common so PhET may reevaluate this position.

For Flash sims, right-to-left writing direction is problematic. Flash relies on the operating system to handle writing direction, so the writing direction will be correct only if the locale of the user's computer matches the sim's locale. But this conflicts with delivery requirements; sims are supposed to run in a locale that is independent of the user's computer. We are investigating solutions to this problem.

Strings that are common to many simulations are stored in special common-string files. These strings can be translated using the PhET translation Utility, but they cannot be tested because they have no associated simulation. This presents challenges for translators, since they have no context in which to evaluate translations of common strings. It also presents challenges for integrating and testing translations, since submitted translations for common strings must be integrated with all simulations. PhET is still refining the process of translating and integrating common strings.

Features that were originally intended for one simulation often are found to be generally useful, and are then migrated to PhET's common framework. If these features have associated strings, then the translations of those strings also need to be migrated from sim-specific string files to the common-string files. That migration is currently an expensive manual process. PhET is investigating how to automate this process

Two types of strings have proven to by problematic and error-prone for translators: HTML strings and MessageFormat strings. Both types of strings require special knowledge that translators may not have. PhET is investigating adding additional support to the PhET Translation Utility for these types of strings.

While the PhET Translation Utility provides the ability to test translations, it does not detect user-interface layout problems. Layout problems can occur when a translated string is significantly longer or shorter than the original English string. (Figure 3 shows an example.) Longer strings in particular can cause part of the simulation's user-interface to overlap or become unusable. A translator who is not thoroughly familiar with a simulation may not notice these issues.

References

[1] Adams, W.A. et. al., *Making On-line Science Course Materials Easily Translatable and Accessible Worldwide: Challenges and Solutions*, 2009.

[2] http://en.wikipedia.org/wiki/Locale

[3] http://en.wikipedia.org/wiki/List_of_ISO_639-1_codes

[4] http://en.wikipedia.org/wiki/ISO_3166-1_alpha-2

[5] http://en.wikipedia.org/wiki/Internationalization_and_localization

[6] http://unicode.org/standard/standard.html