Autodesk Maya Plug-in Internationalization White Paper

This white paper describes the internationalization and localization features available to programmers developing plug-ins for Autodesk® Maya® software. The paper describes recommended application programming interface (API) techniques for operating in localized user environments, in particular those requiring multi-byte text encodings. It also explains how to adapt plug-ins to provide their user interface elements in one or more alternate languages.

Contents

Autodesk®

Introduction

Autodesk® Maya® software is an internationalized application. This means that the software is capable of supporting alternate languages in its user interface and it is compatible with localized user environments requiring either single-byte or multi-byte text encodings.

Maya runs by default in the English language. When Maya is running in an alternate language, user messages, menus, labels, and other user interface text are displayed with translated values in place of the default English values.

This white paper is mainly of interest to software developers who wish to write plug-ins that are compatible within localized user environments and operate correctly with both single and multi-byte text encodings. It also explains how to adapt plug-in code and scripts to support one or more alternate user interface languages.

Overview of the Maya API Internationalization Architecture

This section gives an overview of the main concepts and features that provide Internationalization and Localization support in the Maya application programming interface (API). Implementation details and examples are found in subsequent sections.

Internationalization and Localization

There are two main processes involved in preparing plug-ins to run in alternate languages and locales; Internationalization and Localization.

Internationalization is the process of changing the executable code and scripts so that they can be localized. In the context of plug-ins, this mainly involves replacing hard-coded user interface strings with string resources that can be changed to match the user's language choice. String handling code may also require changes to work correctly in all language encodings.

Localization is the process of customizing the plug-in to work in a particular language. For a plug-in, this will mainly involve translating the user interface strings and installing them correctly in the plug-in resource path. Once the code is internationalized, it can be localized for one or more target languages.

Single-binary Strategy

The Maya application is internationalized using a single-binary strategy. This means that the same executable code can support different languages. All data that varies between user language selections is external to the software itself and is supplied as separate Localization packages. The software itself does not need to be re-built to support a different user language or to run under different codepage settings. This philosophy extends to the Internationalization features provided in the Maya API.

Locales and Encodings

Localized user environments will have different locale and codepage (encoding) settings. Maya automatically determines the user language preference and the associated locale and codepage. The manner in which these are determined from user preferences differs across hardware platforms, and is described in the Maya Documentation.

To see what user interface language and codepage Maya is using, type the following commands:

```
about -uiLanguage;
about -codeset;
```

To a large extent, proper use of the Maya API classes and methods insulates plug-in code from the locale settings and their implications. It is important, however, to understand that these settings affect how string data is interpreted and how external interfaces will behave, especially when dealing with multi-byte locales.

Localized Scripts

Maya Embedded Language (MEL) scripts are often provided with a plug-in to create custom user interfaces.

MEL scripts are simply text files that are interpreted using the code page of the current locale. They can be edited to use localized string values for user interface (UI) labels, user messages, etc. In the simplest case, when a plug-in is to be provided in one language only, it is often possible to provide the custom user interfaces with localized MEL scripts.

If a plug-in is to be localized into multiple languages, the scripts should be adapted to use String Resources, which can be set to alternate values at run time based on the user's language choice. String resources are described below.

Localized Strings

The Maya API has its own string class, `MString`, which is used throughout the API to pass textual data between methods. `MString` provides methods for manipulating and accessing the string data, which to a large extent insulate the code from any concern about the underlying representation of the string.

The application locale affects the behavior of `MString` methods that accept or return string data in character (`char *`) buffers, since the correct data encoding must be used to interpret the character stream. In some cases adaptations to plug-in code accessing or manipulating character buffers will be required so that the strings are treated correctly, especially in multi-byte environments. More information about using `MString` and its methods in a localized environment can be found under String Handling below.

String Resources

The Maya API has been extended to provide a string resource mechanism for plug-ins. This allows plug-ins to add strings to the Maya String Catalog. Localized values for these strings can be provided for use when Maya is running in an alternate language.

Plug-in string resources can be used in both C++ code and MEL scripts. The plug-in resource architecture is designed so that the default (English) string values are specified directly in the plug-in source code (C++ or script). The default values are "built-in" and are always available.

Localized resource values are provided in external files that are read and loaded into the Maya string catalog when the plug-in is loaded. Adding localized resources does not require rebuilding the plug-in executable.

Localized resources are optional. One advantage to this approach is that plug-ins can be adapted to use string resources but the Localization (or translation) of the resources can be deferred until a later time. The localized resources will only be used if they are available and match the user's language choice. If localized resources are not found for any reason the plug-in will run as before with the default (built-in) string values.

Internationalization Process

Internationalization enables the plug-in to support localized strings in its user interface and to operate correctly in localized environments. The process of localizing the string resources is a separate set of steps outlined later in this document. Localization can take place any time

after the plug-in has been Internationalized. The plug-in will continue to work using the default resource values until translations are available.

The following steps are involved in Internationalization:

1. determine what aspects of the plug-in will require Localization
2. identify the user interface strings that need to change when the plug-in is running in another language. The strings may be located in C++ code or MEL scripts. Modify the code and scripts to use string resources in place of hard-coded strings
3. add string registration calls to the plug-in initialization sequence
4. review string handling code and make modifications as required to properly handle strings in both single and multi-byte encodings

Assessing Requirements

Before doing the work to internationalize a plug-in, it is useful to assess what the actual requirements are. Not all aspects of plug-in Internationalization may be necessary or appropriate for each situation.

The most important consideration is to determine the number of target languages the plug-in needs to support. If a plug-in is only targeted at a single language (e.g., only Japanese) it may be possible to supply the custom UI for the plug-in as a localized script instead of using string resources.

String resources are required if the plug-in meets one or more of the following criteria:

- the plug-in is to support more than one user interface language (e.g., English and Japanese)
- the plug-in issues user messages from C++ code (e.g. calls to `displayError()`)
- the plug-in creates nodes which are to be displayed in the UI with localized node and attribute name labels

Another consideration is the type of string handling and I/O the plug-in code performs. Plug-ins with little or no string manipulation or file handling should require few changes, while others will need to be reviewed to ensure they are dealing properly with multi-byte characters.

Defining and Registering String Resources

This section will discuss how string resources are used in C++ and MEL scripts, and how to register them during plug-in initialization.

String Resource Keys

Each string resource in the Maya String Catalog is identified by a unique key. For plug-ins, the key consists of an ordered pair of strings. The first element of the pair is the plug-in name, which will be the same for all strings used by the plug-in. The second part of the key is a unique identifier for the string being defined.

For example, string resource keys used by the `closestPointOnCurve` plug-in will have the form:

```
("closestPointOnCurve", "stringId1");
("closestPointOnCurve", "stringId2");
("closestPointOnCurve", "stringId3");
```

String Resources in C++

The `MStringResourceId` and `MStringResource` classes are used to define and access string resources in the plug-in C++ code.

The `MStringResourceId` constructor accepts three arguments, the two elements used to form the resource key, and the default value of the resource string. In the example below the plug-in name is "`closestPointOnCurve`", and the unique user-defined key being given to this string is "`kNoValidObject`".

```
MStringResourceId invalidObject("closestPointOnCurve", "kNoValidObject",
"A curve or its transform node must be specified as a command argument,
or using your current selection.");
```

Using #define statements to associate the `MStringResourceId` with a constant is helpful to provide a single point of definition for each resource. Typically the `MStringResourceId` declarations for the plug-in will be grouped together in a header file that is shared between the C++ modules that require it.

```
#define kPluginId  "closestPointOnCurve"
#define kNoValidObject MStringResourceId(kPluginId, "kNoValidObject", \
"A curve or its transform node must be specified as a command argument, or
using your current selection.")
```

The `MStringResource::getString` method is used to look up the catalog entry when the string is needed in the code. The catalog lookup will return either the default value or localized string value (if it is available). Once the string is loaded it can be used like any other MString.

```
MStatus stat;
MString msg = MStringResource::getString(kNoValidObject, stat);
displayError(msg);
```

A string resource cannot be accessed until it has been registered. This is done by calling `MstringResource::registerString`. The registration steps are described below under "String Resource Registration".

String Resources in MEL Scripts

MEL scripts can use string resources in a similar manner to the C++ code.

To use a string resource, the value is retrieved using the `getPluginResource` command. The arguments passed are the two elements of the string resource key.

```
string $titleStr = getPluginResource("closestPointOnCurve", "kAETitle");
editorTemplate -beginScrollLayout;
editorTemplate -beginLayout $titleStr -collapse 0;
```

MEL resources are registered using the `registerPluginResource` command. The registration process is described below.

String Resource Registration

This section describes the process of registering string resources in the plug-in. All string registration is done during plug-in initialization.

Each string resource must be registered before it can be used. The registration step ensures that the resource's default value is loaded into the string catalog. After the default values are registered the plug-in writer must add a call which will load the localized resource values. When localized resource values are available for the language Maya is running in, the localized values will override the default values.

The main registration method is `MFnPlugin::registerUIStrings`. A call to this routine is added to the plug-in's `initializePlugin()` function. It should be placed early in the initialization sequence since the string resources will not be available until it is called, and some of the other initialization methods may require them.

The `MFnPlugin::registerUIStrings` function takes two arguments. The first argument is the name of a procedure which will register the strings used in the C++ code. The second argument is the name of a script that will register the string resources used in MEL scripts.

```
   // Register string resources used in the code and scripts

   status = plugin.registerUIStrings(registerMStringResources,
"closestPointOnCurveInitStrings");
   if (!status)
   {
      status.perror("registerUIStrings");
      return status;
   }
```

The C++ routine `registerMStringResources` referenced in this call registers each `MStringResourceId` used by the C++ code.

```
// Register all strings used by the plugin C++ source
static MStatus registerMStringResources(void)
{
        MStringResource::registerString(kNoValidObject);
        // other resources would go here
        return MS::kSuccess;
}
```

Note: this example is using a constant previously defined in a header file:

```
#define kPluginId  "closestPointOnCurve"
#define kNoValidObject MStringResourceId(kPluginId, "kNoValidObject", \
"A curve or its transform node must be specified as a command argument, or
using your current selection.")
```

The second argument is the name of a script which initializes all resources used by the plug-in's MEL scripts. Each resource is registered with a call to `registerPluginResources`.

The initialization script serves a dual purpose. In addition to the MEL resource registration, it also contains the logic to load language-dependent resources for the plug-in. The routine `loadPluginLanguageResources` takes the name of a resource file that will contain the localized version of the plug-in string resources. More details creating and installing the localized resource file are found under Localization Process below.

```
global proc closestPointOnCurveInitStrings()
{
    // Register script resources
    registerPluginResource("closestPointOnCurve", "kAETitle",
                           "Closest Point On Curve Attributes");
    registerPluginResource("closestPointOnCurve", "kInputCurve",
                           "Input Curve");
    registerPluginResource("closestPointOnCurve", "kResults",
                           "Results");
```

```
    // Load any localized resources
    loadPluginLanguageResources("closestPointOnCurve",
"closestPointOnCurve.pres.mel");
}
```

Once the registration sequence is complete the strings are available in the Maya Catalog and can be retrieved using `MString::getStringResource` (C++) or `getPluginResource` (MEL). If localized values for the resources were located by `loadPluginLanguageResources` they will be returned from the catalog instead of the default values.

String Handling

Plug-in writers using the `MString` class are largely insulated from locale-dependent changes, but some string handling code may require changes to operate correctly in both single and multi-byte environments. This section mainly focuses on issues as they relate to the use of `MString` in a localized environment, but many of the problems described apply in to character handling in general.

Encoding

The `MString` class operates under the assumption that by default, character data in `char*` form is encoded in the codeset of the locale. This is a natural extension of the existing functionality of the class and in many cases an existing plug-in will continue to work without changes in a localized environment. New methods have been added to explicitly assign or access the string using UTF-8 and wide character formats which are commonly used in internationalized applications.

String Length and Position Values

The most common problem when dealing with localized text is the correct interpretation of string length. In multi-byte environments, the character (`char *`) representation of the string will use one or more bytes to represent each character in the string. This means that the string's storage length in bytes does not necessarily correspond to the number of individual characters in the string and code using this assumption may not behave as expected. For backwards compatibility, the `MString::length` method will continue to return the number of bytes in the character buffer. A new method `MString::numChars` can be used instead when it is necessary to determine the number of individual characters in the string.

The interpretation of positional indexes into the string data is similarly problematic in a multi-byte environment (for example when using the `MString::substring` method). See the “New MString Methods” section below for details on what has been added to deal correctly with multi-byte strings.

Formatting Message Strings

User message strings are often built by concatenating together multiple strings or variables. This technique is not appropriate for strings that will be localized, since the context and placement of the strings may need to change for another language.

The `MString::format` method or MEL `format` command should be used to format the string. Format allows positional arguments to be correctly placed in context when the string is translated.

The following example shows an original block of code which creates an error message string containing the name of a file using string concatenation:

```
MString filename;
MString msg;
msg = “The file “;
msg += filename;
msg += “ cannot be opened”;
```

```
displayError(msg);
```

The following replacement code shows how this would be done correctly in an internationalized application. The message is being created using a resource string and the `MString::format` command:

```
#define kOpenError MStringResourceId("myPlugin", "kOpenError",
"File ^1s cannot be opened");

MString filename;
MString msgFmt = MStringResource::getString(kOpenError,status);
MString msg;
msg.format(msgFmt, filename);
displayError(msg);
```

MEL scripts can make use of the MEL `format` command in a similar manner.

New MString Methods

The following table lists new methods that have been added to `MString` to support Internationalization. See the `MString` class documentation for more details about each method, as well as notes about the behavior of existing methods within a localized environment.

New MString Method	Notes
`MString::numChars`	This routine returns the number of characters in the string. This does not necessarily correspond to the number of bytes in the string, or the value returned by `MString::length`.

`MStatus MString::setUTF8( const char * utf8String )`
`const char * MString::asUTF8()`
`const char * MString::asUTF8(int *length)`
	These methods assign or access the string value as a UTF-8 encoded character string.

`Mstring::MString(wchar_t *str)`
`MString::MString(wchar_ *str, int length)`
`MStatus MString::setWChar(wchar_t *str, int length)`
`MStatus MString::setWChar(wchar_t *str, int length)`
`const wchar_t* MString::asWChar()`
`const wchar_t* MString::asWChar(int length)`
	These methods allow the string's value to be set or accessed using wide character values. Note: the use of wide character representation is not recommended for persistent storage; use of a portable format, such as UTF-8, is recommended instead.

`int MString::indexW(char c) const`
`int MString::indexW(wchar_t c) const`
`int MString::rindexW(char c) const`
`int MString::rindexW(wchar_t c) const`
	These routines are multi-byte compatible versions of `MString::index` and `MString::rindex` respectively and return character-based position values. Use of these routines for internationalized plug-ins is recommended. See also `MString::substringW`.

`MString MString MString::substringW(int start, int end)`	This is a mult-byte compatible version of `MString::substring` which accepts character-based position values (such as those returned by `MString::indexW` and `MString::rindexW`).
`MStatus MString::split(wchar_t c, MStringArray& array) const`	This version of `MString::split` accepts a wide-character value as the delimiter.
`MStatus MString::format( const MString &fmt, const MString &arg1, const MString &arg2, ... ,const MString &arg10 )`	This routine provides string formatting capabilities, using a format specifier and up to 10 positional arguments.

Localization Process

After the plug-in has been internationalized, string resources can be extracted and sent for translation. It is important to note that the process of Localization can be deferred. The plug-in will continue to run with its default resource values if the localized versions are not available.

Plug-in Localization involves the following steps:

1. Resource Extraction: a master resource file containing all localizable strings from the plug-in is generated.
2. Translation: a translated version of the strings is prepared.
3. Installation: the translated resource file is installed into the correct language area on the plug-in resource path.

Resource Extraction

The utility script `pluginResourceUtil` is used to generate a master list of all registered resources for the plug-in.

To run the utility, the name of the plug-in and the name of the output file to generate is specified. The plug-in must be loaded for the resource extraction process to take place; the utility will load the plug-in if it is not already loaded. The extraction process must also be done while Maya is running in the default (English) language. See the Maya documentation for how to override the Maya language setting.

```
pluginResourceUtil("closestPointOnCurve", "c:/extracted/closestPointOnCurve.pres.mel");
```

The output from the resource extraction process is a file containing a list of commands to set the resources to new values. This master file is not used by the plug-in (the default resource values contained in this file are already available to the plug-in without it). The file is used as the master version of the strings that need to be translated. Only the translated versions of the file are provided with the plug-in.

The complete output file `closestPointOnCurve.pres.mel` is shown in Appendix A, a portion of the file is shown below:

```
// File closestPointOnCurve.pres.mel

// Resources for Plug-in: closestPointOnCurve
//
// ----------------------------
// Registered string resources:
// ----------------------------
setPluginResource( "closestPointOnCurve", "kAETitle", "Closest Point On Curve
Attributes");
setPluginResource( "closestPointOnCurve", "kInputCurve", "Input Curve");
setPluginResource( "closestPointOnCurve", "kInvalidType", "Object ^1s has invalid
type.  Only a curve or its transform can be specified.");
setPluginResource( "closestPointOnCurve", "kNoQueryFlag", "You must specify AT
LEAST ONE queryable flag in query mode.  Use the `help` command to list all
available flags.");
setPluginResource( "closestPointOnCurve", "kNoValidObject", "A curve or its
transform node must be specified as a command argument, or using your current
selection.");
setPluginResource( "closestPointOnCurve", "kResults", "Results");
```

The generated file contains an entry for each string resource registered by the plug-in (both C++ and MEL resources). Additionally, if the plug-in has registered nodes, entries for the standardized node and attribute UI display string resources that are used by Maya will be automatically generated (there is no additional registration required for these resources other than correctly registering the node).

Translation

The master resource file is translated into another language by editing the string values for each entry in the resource file.

Encoding of Resource Files

Since the resource file is a MEL text file its encoding must be appropriate for the locale and platform that it will run on. For example, Japanese translations on Windows platforms will expect CP932, while on Mac OS X they should be UTF-8. Conversion utilities such as `iconv` can be used for converting file formats if required.

Installation of Localized Resources

The translated resource file must be installed in the correct language-dependent location, so that it will be loaded at runtime.

The resource file will have the same name for each language it is translated to, and the directory it is located in will determine the language it is associated with. The resource file name is passed to `loadPluginLanguageResources` in the string initialization script.

In the example below, the resource file is named `closestPointOnCurve.pres.mel`.

```
// Load any localized resources
    loadPluginLanguageResources("closestPointOnCurve",
"closestPointOnCurve.pres.mel");
```

The `loadPluginLanguageResources` routine will search for the resource file along the MAYA_PLUG_IN_RESOURCES_PATH.

Typically plug-ins are installed as modules and their files are installed within a standardized directory structure along the MAYA_MODULE_PATH. The MAYA_PLUG_IN_RESOURCES_PATH will be initialized to include language-specific resource directory entries for each module.

A sample directory hierarchy for a plug-in module is shown below. The resources area contains subdirectories for each available Localization. The Japanese resources would be installed into the `resources/ja_JP` subdirectory. When Maya is running in Japanese this directory will be added to the MAYA_PLUG_IN_RESOURCES_PATH.

`loadPluginLanguageResources` will not generate an error at runtime if the resource file is not found. The plug-in will simply continue to operate with the default resource values.

The example below shows a sample module hierarchy and the location of Japanese resources for the closestPointOnCurve plug-in.

```
/SampleModule
/SampleModule/data
/SampleModule/docs
/SampleModule/icons
/SampleModule//modules
/SampleModule/plug-ins
/SampleModule/python
/SampleModule/resources
/SampleModule/resources/ja_JP
```

```
/SampleModule/resources/ja_JP/closestPointOnCurve.pres.mel
/SampleModule/scripts
```

Maintenance

The resource file extraction, translation and installation steps will need to be repeated if the plug-in string resources are modified, to keep the localized versions synchronized with the master versions.

Appendix A: Example

The following source code taken from the closestPointOnCurve plug-in demonstrates how it was modified to support localized user interface strings. Relevant changes have been **highlighted**. The complete set of source code for this plug-in is available as part of the Maya Bonus Tools.

closestPointOnCurvePlugin.cpp:

When the plug-in is initialized, a call to MfnPlugin::registerUIStrings is made to register the strings used by the C++ code, and to invoke the MEL command closestPointOnCurveInitStrings which registers MEL string resources and loads localized values. The closestPointOnCurveStrings.h header file was created to provide a single point of definition for the C++ string resources used by the plug-in.

```
// File: closestPointOnCurveStrings.cpp

// HEADER FILES:
#include "closestPointOnCurveCmd.h"
#include "closestPointOnCurveNode.h"
#include "closestPointOnCurveStrings.h"
#include <maya/MFnPlugin.h>

// Register all strings used by the plugin C++ source
static MStatus registerMStringResources(void)
{
      MStringResource::registerString(kNoValidObject);
      MStringResource::registerString(kInvalidType);
      MStringResource::registerString(kNoQueryFlag);
      return MS::kSuccess;
}

// INITIALIZES THE PLUGIN BY REGISTERING COMMAND AND NODE:

MStatus initializePlugin(MObject obj)
{
   MStatus status;
   MFnPlugin plugin(obj, PLUGIN_COMPANY, "4.0", "Any");


   // Register string resources used in the code and scripts
   // This is done first, so the strings are available.
   status = plugin.registerUIStrings(registerMStringResources,
"closestPointOnCurveInitStrings");
   if (!status)
   {
      status.perror("registerUIStrings");
      return status;
   }


   status = plugin.registerCommand("closestPointOnCurve",
closestPointOnCurveCommand::creator, closestPointOnCurveCommand::newSyntax);
```

```
    if (!status)
    {
        status.perror("registerCommand");
        return status;
    }

    status = plugin.registerNode("closestPointOnCurve",
closestPointOnCurveNode::id, closestPointOnCurveNode::creator,
closestPointOnCurveNode::initialize);

    if (!status)
    {
        status.perror("registerNode");
        return status;
    }
    return status;
}
```

closestPointOnCurveStrings.h:

This header file was added to define the string resources used in the plug-in's C++ modules. When the resources are accessed in more than one source file this type of approach is recommended for providing a common set of definitions for the `MStringResourceId` values.

```
// File: closestPointOnCurveStrings.h

// MAYA HEADER FILES:
#include <maya/MStringResource.h>
#include <maya/MStringResourceId.h>

// MStringResourceIds contain plugin id, unique resource id for
// each string and the default value for the string.

#define kPluginId  "closestPointOnCurve"

#define kNoValidObject MStringResourceId(kPluginId, "kNoValidObject", \
"A curve or its transform node must be specified as a command argument, or using
your current selection.")

#define kInvalidType  MStringResourceId(kPluginId, "kInvalidType", \
"Object ^1s has invalid type.  Only a curve or its transform can be specified.")

#define kNoQueryFlag  MStringResourceId(kPluginId, "kNoQueryFlag", \
"You must specify AT LEAST ONE queryable flag in query mode.  Use the `help`
command to list all available flags.")
```

closestPointOnCurveCmd.cpp:

This file previously used hard-coded strings to display error messages. The hard-coded strings were redefined as `MStringResourceId` objects in closestPointOnCurveStrings.h. To use the string resources, a call to `MStringResource::getString` is made to retrieve the current value of the resource.

This code also illustrates the use of the `MString::format` method for inserting variable arguments into message strings. Using format for constructing message strings is recommended for internationalized code instead of string concatenation. Formatted strings allow positional arguments to be correctly placed in the correct context and position when the string is translated to another language.

```
// FILE: closestPointOnCurveCmd.cpp

// HEADER FILES:
#include "closestPointOnCurveCmd.h"
#include "closestTangentUAndDistance.h"
#include "closestPointOnCurveStrings.h"



// COMPUTING THE OUTPUT VALUES FOR THE CLOSEST POSITION, NORMAL, TANGENT,
// PARAMETER-U AND DISTANCE, OR CREATING A "closestPointOnCurve" NODE:

MStatus closestPointOnCurveCommand::redoIt()
{
   // DOUBLE-CHECK TO MAKE SURE THERE'S A SPECIFIED OBJECT TO EVALUATE ON:
   if (sList.length() == 0)
   {
      MStatus stat;
      MString msg = MStringResource::getString(kNoValidObject, stat);
      displayError(msg);
      return MStatus::kFailure;
   }

   // RETRIEVE THE SPECIFIED OBJECT AS A DAGPATH:

   MDagPath curveDagPath;
   sList.getDagPath(0, curveDagPath);

   // CHECK FOR INVALID NODE-TYPE INPUT WHEN SPECIFIED/SELECTED
   // NODE IS *NOT* A "CURVE" NOR "CURVE TRANSFORM":

   if (!curveDagPath.node().hasFn(MFn::kNurbsCurve) &&
       !(curveDagPath.node().hasFn(MFn::kTransform)
       && curveDagPath.hasFn(MFn::kNurbsCurve)))
   {
      MStatus stat;
      MString msg;
      // Use format to place variable string into message
      MString msgFmt = MStringResource::getString(kInvalidType, stat);
      MStringArray selectionStrings;
      sList.getSelectionStrings(0, selectionStrings);
      msg.format(msgFmt, selectionStrings[0]);
      displayError(msg);
      return MStatus::kFailure;
   }

   // WHEN COMMAND *NOT* IN "QUERY MODE" (I.E. "CREATION MODE"), CREATE AND
   // CONNECT A "closestPointOnCurve" NODE AND RETURN ITS NODE NAME:

   if (!queryFlagSet)
```

```
{
   // CREATE THE NODE:

   MFnDependencyNode depNodeFn;
   if (closestPointOnCurveNodeName == "")
      depNodeFn.create("closestPointOnCurve");
   else
      depNodeFn.create("closestPointOnCurve", closestPointOnCurveNodeName);
   closestPointOnCurveNodeName = depNodeFn.name();

   // SET THE ".inPosition" ATTRIBUTE, IF SPECIFIED IN THE COMMAND:

   if (inPositionFlagSet)
   {
      MPlug inPositionXPlug = depNodeFn.findPlug("inPositionX");
      inPositionXPlug.setValue(inPosition.x);
      MPlug inPositionYPlug = depNodeFn.findPlug("inPositionY");
      inPositionYPlug.setValue(inPosition.y);
      MPlug inPositionZPlug = depNodeFn.findPlug("inPositionZ");
      inPositionZPlug.setValue(inPosition.z);
   }

   // MAKE SOME ADJUSTMENTS WHEN THE SPECIFIED NODE IS A
   // "TRANSFORM" OF A CURVE SHAPE:

   unsigned instanceNumber=0;
   if (curveDagPath.node().hasFn(MFn::kTransform))
   {
      // EXTEND THE DAGPATH TO ITS CURVE "SHAPE" NODE:
      curveDagPath.extendToShape();

      // TRANSFORMS ARE *NOT* NECESSARILY THE "FIRST" INSTANCE
      // TRANSFORM OF A CURVE SHAPE:
      instanceNumber = curveDagPath.instanceNumber();
   }

   // CONNECT THE NODES:

   MPlug worldCurvePlug, inCurvePlug;
   inCurvePlug = depNodeFn.findPlug("inCurve");
   depNodeFn.setObject(curveDagPath.node());
   worldCurvePlug = depNodeFn.findPlug("worldSpace");
   worldCurvePlug = worldCurvePlug.elementByLogicalIndex(instanceNumber);
   MDGModifier dgModifier;
   dgModifier.connect(worldCurvePlug, inCurvePlug);
   dgModifier.doIt();

   // SET COMMAND RESULT TO BE NEW NODE'S NAME, AND RETURN:
   setResult(closestPointOnCurveNodeName);
   return MStatus::kSuccess;
}

// OTHERWISE, WE'RE IN THE COMMAND'S "QUERY MODE":

else
{
```

```
        // COMPUTE THE CLOSEST POSITION, NORMAL, TANGENT, PARAMETER-U
        // AND DISTANCE, USING THE *FIRST* INSTANCE TRANSFORM WHEN CURVE
        // IS SPECIFIED AS A "SHAPE":

        MPoint position;
        MVector normal, tangent;
        double paramU, distance;
        closestTangentUAndDistance(curveDagPath, inPosition, position,
                normal, tangent, paramU, distance);

        // WHEN NO QUERYABLE FLAG IS SPECIFIED, INDICATE AN ERROR:

        if (!positionFlagSet && !normalFlagSet && !tangentFlagSet && !paramUFlagSet
&& !distanceFlagSet)
        {
        MStatus stat;
        MString msg = MStringResource::getString(kNoQueryFlag, stat);
        displayError(msg);
        return MStatus::kFailure;
       }

        // WHEN JUST THE "DISTANCE" IS QUERIED, RETURN A SINGLE
        // "FLOAT" INSTEAD OF AN ENTIRE FLOAT ARRAY FROM THE COMMAND:

        else if (distanceFlagSet && !(positionFlagSet || normalFlagSet ||
tangentFlagSet || paramUFlagSet))
           setResult(distance);

        // WHEN JUST THE "PARAMETER-U" IS QUERIED, RETURN A
        // SINGLE "FLOAT" INSTEAD OF AN ENTIRE FLOAT ARRAY FROM THE COMMAND:

        else if (paramUFlagSet && !(positionFlagSet || normalFlagSet ||
tangentFlagSet || distanceFlagSet))
           setResult(paramU);

        // OTHERWISE, SET THE RETURN VALUE OF THE COMMAND'S RESULT TO
        // A "COMPOSITE ARRAY OF FLOATS":

        else
        {
           // HOLDS FLOAT ARRAY RESULT:

           MDoubleArray floatArrayResult;

           // APPEND THE RESULTS OF THE CLOSEST POSITION, NORMAL,
           // TANGENT, PARAMETER-U AND DISTANCE VALUES TO THE FLOAT ARRAY RESULT:

           if (positionFlagSet)
           {
              floatArrayResult.append(position.x);
              floatArrayResult.append(position.y);
              floatArrayResult.append(position.z);
           }

           if (normalFlagSet)
           {
              floatArrayResult.append(normal.x);
              floatArrayResult.append(normal.y);
```

```
            floatArrayResult.append(normal.z);
        }

        if (tangentFlagSet)
        {
            floatArrayResult.append(tangent.x);
            floatArrayResult.append(tangent.y);
            floatArrayResult.append(tangent.z);
        }

        if (paramUFlagSet)
            floatArrayResult.append(paramU);
        if (distanceFlagSet)
            floatArrayResult.append(distance);

        // FINALLY, SET THE COMMAND'S RESULT:

        setResult(floatArrayResult);
      }
      return MStatus::kSuccess;
   }
}
```

closestPointOnCurveCmdInitStrings.mel:

This file is the string initialization script referenced in the call to `MfnPlugin::registerStringResources`. It has a dual purpose:

1. It registers any string resources used by the plug-in MEL scripts. The strings registered in this example are used in `AEclosestPointOnCurveTemplate.mel`.
2. It calls `loadPluginLanguageResources` with the name of the file containing the localized string values for this plug-in. In this example, a file named `closestPointOnCurve.pres.mel` will be loaded if it is located in the appropriate language-specific location expected by `loadPluginLanguageResources`.

```
// FILE: closestPointOnCurveInitStrings.mel
// DESCRIPTION: Register script resources and load localized resources
//              for the "closestPointOnCurve" plugin
global proc closestPointOnCurveInitStrings()
{
    // Register script resources
    registerPluginResource("closestPointOnCurve", "kAETitle",
                           "Closest Point On Curve Attributes");
    registerPluginResource("closestPointOnCurve", "kInputCurve",
                           "Input Curve");
    registerPluginResource("closestPointOnCurve", "kResults",
                           "Results");

    // Load any localized resources
    loadPluginLanguageResources("closestPointOnCurve",
"closestPointOnCurve.pres.mel");
}
```

AEclosestPointOnCurveTemplate.mel:

This file defines the custom attribute editor setup for the closestPointOnCurve node which is created by this plug-in. The code has been modified to use string resources in place of hard-coded strings for the attribute editor labels.

Note: the attribute names themselves can also be displayed in translated form, but the programmer does not need to designate string resources manually for each attribute. All attributes defined by the plug-in will have attributeNiceName resources automatically generated in the extracted resource file.

```
// FILE: AEclosestPointOnCurveTemplate.mel

global proc AEclosestPointOnCurveTemplate(string $nodeName)
{
      string $titleStr = getPluginResource("closestPointOnCurve", "kAETitle");
      string $inputCurveLabel = getPluginResource("closestPointOnCurve",
                        "kInputCurve");
      string $resultLabel = getPluginResource("closestPointOnCurve", "kResults");

      editorTemplate -beginScrollLayout;
      editorTemplate -beginLayout $titleStr -collapse 0;
         editorTemplate -callCustom ( "AEinputNew \""+ $inputCurveLabel +"\"" )
               ( "AEinputReplace \"" + $inputCurveLabel + "\"" )
                  "inCurve";
         editorTemplate -addControl "inPosition";
         editorTemplate -beginLayout $resultLabel;
            editorTemplate -addControl "position";
            editorTemplate -addControl "normal";
            editorTemplate -addControl "tangent";
            editorTemplate -addControl "paramU";
            editorTemplate -addControl "distance";
         editorTemplate -endLayout;
      editorTemplate -endLayout;
      editorTemplate -suppress "inCurve";
      AEabstractBaseCreateTemplate $nodeName;
      editorTemplate -addExtraControls;
   editorTemplate -endScrollLayout;
}
```

closestPointOnCurve.pres.mel:

This file contains the extracted string resources for the closestPointOnCurve plug-in. It was generated using the utility script `pluginResourceUtil`. All registered strings from the C++ code and MEL scripts are extracted to the file along with their default values. Also included are node and attribute nice name values for each node registered by the plug-in, which are generated automatically by the utility; (there is no manual registration required for these node and attribute resources).

It is important to note that this original extracted file containing the default values is not required by the plug-in when it is running in English. All string resources have their default value built directly into the plug-in itself. The extracted file is used as the master list of resources that are to be translated into other languages. When the translated versions are available, the files are placed in the appropriate language-specific resources sub-directory along the MAYA_PLUG_IN_RESOURCE_PATH.

```
// File closestPointOnCurve.pres.mel

// Resources for Plug-in: closestPointOnCurve
//
// ----------------------------
// Registered string resources:
// ----------------------------
setPluginResource( "closestPointOnCurve", "kAETitle", "Closest Point On Curve
Attributes");
setPluginResource( "closestPointOnCurve", "kInputCurve", "Input Curve");
setPluginResource( "closestPointOnCurve", "kInvalidType", "Object ^1s has invalid
type.  Only a curve or its transform can be specified.");
setPluginResource( "closestPointOnCurve", "kNoQueryFlag", "You must specify AT
LEAST ONE queryable flag in query mode.  Use the `help` command to list all
available flags.");
setPluginResource( "closestPointOnCurve", "kNoValidObject", "A curve or its
transform node must be specified as a command argument, or using your current
selection.");
setPluginResource( "closestPointOnCurve", "kResults", "Results");

//
// ---------------------------
// Registered node resources:
// ---------------------------
//
// Node: closestPointOnCurve
//
setNodeNiceNameResource( "closestPointOnCurve", "Closest Point On Curve" );
setAttrNiceNameResource( "closestPointOnCurve", "ic", "In Curve" );
setAttrNiceNameResource( "closestPointOnCurve", "ip", "In Position" );
setAttrNiceNameResource( "closestPointOnCurve", "ipx", "In Position X" );
setAttrNiceNameResource( "closestPointOnCurve", "ipy", "In Position Y" );
setAttrNiceNameResource( "closestPointOnCurve", "ipz", "In Position Z" );
setAttrNiceNameResource( "closestPointOnCurve", "p", "Position" );
setAttrNiceNameResource( "closestPointOnCurve", "px", "Position X" );
setAttrNiceNameResource( "closestPointOnCurve", "py", "Position Y" );
setAttrNiceNameResource( "closestPointOnCurve", "pz", "Position Z" );
setAttrNiceNameResource( "closestPointOnCurve", "n", "Normal" );
setAttrNiceNameResource( "closestPointOnCurve", "nx", "Normal X" );
setAttrNiceNameResource( "closestPointOnCurve", "ny", "Normal Y" );
setAttrNiceNameResource( "closestPointOnCurve", "nz", "Normal Z" );
setAttrNiceNameResource( "closestPointOnCurve", "t", "Tangent" );
setAttrNiceNameResource( "closestPointOnCurve", "tx", "Tangent X" );
setAttrNiceNameResource( "closestPointOnCurve", "ty", "Tangent Y" );
setAttrNiceNameResource( "closestPointOnCurve", "tz", "Tangent Z" );
setAttrNiceNameResource( "closestPointOnCurve", "u", "Param U" );
setAttrNiceNameResource( "closestPointOnCurve", "d", "Distance" );
```

The translated version of the resource file for Japanese is shown below. The closestPointOnCurve plug-in is a plug-in module, which is installed within a standard module directory structure defined along the MAYA_MODULE_PATH. The Japanese version of the `closestPointOnCurve.pres.mel` file is placed in the `resources/ja_JP` directory.

```
// Resources for Plug-in: closestPointOnCurve
//
// ----------------------------
```

```
// Registered string resources:
// ----------------------------
setPluginResource( "closestPointOnCurve", "kAETitle", "カーブ上の最近接ポイント
アトリビュート");
setPluginResource( "closestPointOnCurve", "kInputCurve", "入力カーブ");
setPluginResource( "closestPointOnCurve", "kInvalidType", "オブジェクト ^1s
は無効なタイプです。 カーブまたはトランスフォームのみ指定できます。");
setPluginResource( "closestPointOnCurve", "kNoQueryFlag", "最低 1
つの照会可能なフラグまたは照会モードを指定する必要があります。 help
コマンドを使用して利用可能なすべてのフラグをリストします。");
setPluginResource( "closestPointOnCurve", "kNoValidObject", "1
つのカーブまたはトランスフォーム
ノードをコマンド引数として、またはカレントの選択項目を使用して指定する必要があります。");
setPluginResource( "closestPointOnCurve", "kResults", "結果");
//
// --------------------------
// Registered node resources:
// --------------------------
//
// Node: closestPointOnCurve
//
setNodeNiceNameResource( "closestPointOnCurve", "カーブ上の最近接ポイント" );
setAttrNiceNameResource( "closestPointOnCurve", "ic", "入力カーブ" );
setAttrNiceNameResource( "closestPointOnCurve", "ip", "入力位置" );
setAttrNiceNameResource( "closestPointOnCurve", "ipx", "入力位置 X" );
setAttrNiceNameResource( "closestPointOnCurve", "ipy", "入力位置 Y" );
setAttrNiceNameResource( "closestPointOnCurve", "ipz", "入力位置 Z" );
setAttrNiceNameResource( "closestPointOnCurve", "p", "位置" );
setAttrNiceNameResource( "closestPointOnCurve", "px", "位置 X" );
setAttrNiceNameResource( "closestPointOnCurve", "py", "位置 Y" );
setAttrNiceNameResource( "closestPointOnCurve", "pz", "位置 Z" );
setAttrNiceNameResource( "closestPointOnCurve", "n", "法線" );
setAttrNiceNameResource( "closestPointOnCurve", "nx", "法線 X" );
setAttrNiceNameResource( "closestPointOnCurve", "ny", "法線 Y" );
setAttrNiceNameResource( "closestPointOnCurve", "nz", "法線 Z" );
setAttrNiceNameResource( "closestPointOnCurve", "t", "接線" );
setAttrNiceNameResource( "closestPointOnCurve", "tx", "接線 X" );
setAttrNiceNameResource( "closestPointOnCurve", "ty", "接線 Y" );
setAttrNiceNameResource( "closestPointOnCurve", "tz", "接線 Z" );
setAttrNiceNameResource( "closestPointOnCurve", "u", "パラメータ U" );
setAttrNiceNameResource( "closestPointOnCurve", "d", "距離" );
```