The Design and Implementation of Multilingualized Squeak

Yoshiki Ohshima, Kazuhiro Abe

Presented at The International Conference on Creating, Connecting and Collaborating through Computing ("C5") January 2003.

Viewpoints Research Institute, 1025 Westwood Blvd 2nd flr, Los Angeles, CA 90024 t: (310) 208-0524

The Design and Implementation of Multilingualized Squeak

Yoshiki Ohshima
Twin Sun, Inc.
360 N. Sepluveda Blvd. Suite 2055
El Segundo, CA USA 90245
Email: Yoshiki.Ohshima@acm.org

Kazuhiro Abe
ViewPoint Technology, Ltd.
3-8-3-205 Ozenji Nishi, Asao-ku
Kawasaki-shi, Kanagawa JAPAN 215-0017
Email: abee.abe@nifty.ne.jp

Abstract

This paper describes the design and implementation of multilingualization ("m17n") of a dynamic object-oriented environment called Squeak. The goal of this project is to provide a collaborative and late-bound environment where the users can use many different natural languages and characters.

Squeak is a highly portable implementation of a dynamic objects environment and it is a good starting point toward the future collaborative environment. However, its text related classes lack the ability to handle natural languages that require extended character sets such as Arabic, Chinese, Greek, Korean, and Japanese.

We have been implementing the multilingualization extension to Squeak. The extension we wrote can be classified as follows: 1) new character and string representations for extended character sets, 2) keyboard input and the file out of multilingual text mechanism, 3) flexible text composition mechanism, 4) extended font handling mechanisms including dynamic font loading and outline font handling, 5) higher level application changes including a Japanese version of SqueakToys.

The resulting environment has the following characteristics: 1) various natural languages can be used in the same context, 2) the pixels on screen, including the appearance of characters can be completely controlled by the program, 3) decent word processing facility for a mixture of multiple languages, 4) existing Squeak capability, such as remote collaborative mechanism will be integrated with it, 5) small memory footprint requirement.

1 Introduction

The more people who live in the networked and computerized society, the more important the communication tools become. We envision that, in near future, non-technical people including kids will express and exchange their ideas using such tools. They may such ideas collaboratively and communicate them over the network [1].

The network and the tool are getting faster and ready for creating such a collaborative environment. What we need is good software that is usable by people all over the world. Because the people will want to express their ideas in their own natural languages, such software must be capable of not only end-user accessible idea authoring, but also multiple natural languages. In the other words, it has to be a *multilingualized* system.

To achieve this goal, we started from a late-bound, dynamic object-oriented system called Squeak. Squeak is an implementation of a dynamic objects environment. Most notably, its virtual machine (*VM*) is written in itself and has the ability to control every pixel displayed on the screen. Squeak also provides various end-user collaborative facilities. Such functionalities include an end-user programming environment called SqueakToys, a web server, a GUI framework that supports remote collaboration, etc.

However, Squeak has a weakness in its multilingual aspect; Squeak's text handling classes lack the ability to handle natural languages other than English. We think that Squeak will be an ideal environment for network collaboration once it is multilingualized.

There are three issues to consider when implementing a multilingual system. Firstly, the displayed or printed result from the system should be acceptable in terms of the appearance.

Secondly, the system should allow the user to use characters from different languages (scripts) without any burden. For instance, the system should easily support applications such as Arabic-English or Chinese-Japanese electric dictionaries in which different scripts are used together.

Thirdly, the system should be portable among the variety of platforms. A multilingualized system will be used

by people who use different hardware and software. The system should be usable on all those platforms.

We aim to fulfill those three requirement in this project. This is an on-going project, but the system is already being used in several places. This fact proves that we are on the right track toward our goal.

In this paper, we discuss the design and implementation of the multilingualization of Squeak.

The following sections are organized as follows. In section 2, we discuss the overall design goals and the prerequisites for understanding the issues of multilingualization. From section 3 to section 9, we discuss the implementation of the added features. Finally, in section 11, we conclude the paper.

2 Overall Design

In this section, we discuss the overall design issues of the multilingualized Squeak.

2.1 Universal Character Set

In the original Squeak, a character, represented as an instance of Character class, holds an 8-bit quantity ("*octet*"). Obviously, that representation is not sufficient for the extended character set.

What kind of representation do we need for the extended character set? One might think that a 16-bit fixed representation would be enough, but even the "industrial standard", Unicode version 3.2 [2] [3], now defines a character set that needs as many as 21 bits per character.

The plain Unicode has another problem; "*han-unification*". The idea behind han-unification is that the standard disregard the glyph difference of certain Kanji characters and let the implementation choose the actual glyph. This abstraction contradicts the philosophy of Squeak; namely, it becomes impossible to ensure pixel identical execution and layout across the platforms.

On the other hand, Unicode seems to be good enough for scripts other than CJKV (in Unicode terminology, CJKV refers to "Chinese, Japanese, Korean and Vietnamese, that use the Chinese origin characters) unified characters. They are well defined and contain many scripts.

Obviously, we need more than 16-bit for a character. What is the upper limit? Actually we don't have to decide the upper limit. Thanks to the late-bound nature of Squeak, we can always change the internal representation without affecting the other parts of system much, even when what is changed is as basic as the Character class. This decision let us start with a 32-bit word for a character. We have added a kind of "encoding tag" to each character to discriminate the unified characters, and also to switch the underlying font and scanner method implementation.

2.2 Memory Usage

To represent text in any form of extended character set, there must be a character entity that can represent more than an 8-bit quantity and a type of string that can store these characters. One way to adapt this new representation is to change Character and String uniformly so that all instances of Character or String represent this new wide character and string (*uniform approach*). One of the most advanced multilingualized system, Emacs after version 20, uses this approach [4]. Another way is to add new representations and let them co-exist with the exisiting default ones (*mixed approach*).

The uniform wide character representation is cleaner, but takes much space. In original version 3.2 image, The total size of the String subinstances occupy is about 1.5MB. If we use unsatisfying 16-bit uniform representation or 32-bit representation, the image size would grow a few megabytes.

We decided to use the mixed approach. The best representation is selected appropriately and implicitly converted to another representation if necessary. In Smalltalk, this kind of implicit conversion is easy to do. Also, migrating from original Squeak to m17n Squeak is easier this way.

2.3 Text Scanning Performance and Flexibility

In the original Squeak, text scanning and displaying are done by primitives if possible. The character to glyph mapping for a given size is one-to-one, so the program simply can lay the glyphs out from left to right. In the other words, text scanning doesn't need much flexibility.

However, for scanning other scripts, we need more flexibility to cope with the different layout rules in different scripts. We thought that some performance help from a new primitive would be necessary, but it turned out that an all Squeak solution was feasible.

2.4 MacRoman vs. ISO-8859-1

For compatibility with the original Squeak, ideally the definition of the one-octet characters and strings should not be changed. However, to adapt to Unicode, it is cleaner if the first 256 characters are identical with ISO-8859-1 character set, instead of the original Squeak's MacRoman character set.

We decided to modify the upper half of the first 256 characters to be ISO-8859-1 characters. Because the characters in the upper half are not used much, this change was easy. With this change, the Character class represents the ISO-8859-1, which is equivalent to the first 256 characters in Unicode standard.

2.5 Keyboard Input

At a layer from the OS to Squeak level code has to create a character in the extended character set from the user keyboard input sequence. One would imagine that modifying VM to pass multi-octet characters to the Squeak level would be a good solution. However, this is not desirable in two reasons. One is that such VM will be incompatible with the existing VMs installed to many computers. The other is that the encoding of the multi octet character sent from the OS is different from one platform to another.

In the current implementation of m17n Squeak, we decide not to modify the VM. If the OS sends a multi octet character to the Squeak VM, the VM treats the octets as if they are separable characters and sent them to the Squeak level code through the Sensor object. The Sensor then interprets the characters if necessary and generate appropriate character.

2.6 Text File Export and Exchange

The requirement for the file out format is two fold. Firstly, it should be understandable by the other non-Squeak software. Secondly, the identity on the roundtrip conversion from Squeak should be ensured. We don't require that the other way of round trip doesn't have to be identical.

Because the original Squeak's file out format already uses the MacRoman format and uses all 8 bits of the octets in a file out, we need to introduce a mechanism to let the characters in the extended character set co-exist with the original MacRoman characters in the file out. To satisfy this goal, there are three feasible encoding schemes for this external file format.

The first possible way is to adapt the X Compound Text format of X Window System, or "*ctext*" [5]. The upside of ctext is compatibility with existing file outs. The 8-bit characters in the file out remain the same semantics. Also, many existing software can read and write this format at least partially. In fact, Japanese in ctext is essentially identical with the standard internet email format for Japanese [6]. The downside of using ctext is that not all scripts in Unicode have a defined sequencer character.

The second way is to use UTF-8 format for file out. The upside of this format is that all Unicode is representable in UTF-8. One downside is that the CJKV characters need to have an extra language tag for the round trip conversion, but there is no standard encoding scheme for this language tag. Another downside is that the encoding for the upper half of the ISO-8859-1 is now different from the existing file out.

The third way is to mix the above two. The file out actually consists of indivisual "chunks" and each chunk can be in different format. If the chunk should be represented in UTF-8, the program puts a special prefix ("`<utf-8>`") and the string up to the terminator ("`!`") is interpreted as UTF-8.

Table 1. Encoding Tag Assignment

encoding name	encoding tag
Latin1	0
JISX0208	1
GB2312	2
KSX1001	3
JISX0208	4
Japanese (U)	5
Simp'd Cn (U)	6
Korean (U)	7
GB2312	8
Trad. Cn (U)	9
Vietnamese (U)	10
KSX1001	12
...	...
LatinExtended (U)	17
IPA (U)	18
...	...
MusicalSymbols (U)	89
MathAlnumSymbols (U)	90
Tags (U)	91
Generic (U)	255

2.7 Conclusion on Design

Overall, the design choices strive to maximize compatibility, minimize memory footprint, and achieve pixel identical execution across platforms.

The following sections will discuss the details of implementation based on the above design.

3 Character Representation

As written in subsection 2.1, we have added a one word per character representation to the original Squeak. In this section, we describe the details of that implementation.

To represent this extended character, a class named MultiCharacter was added as a subclass of Character. Because the value instance variable of Character is already a SmallInteger, MultiCharacter doesn't have to have any additional instance variable. While it is possible to assign any object to value instance, so far we stick to the positive value range of SmallInteger to avoid large integer arithmetic and confusion with negative value. Because one bit is used for the SmallInteger tag and another is for the sign bit, 30 bit out of 32 bit word is actually available for positive integer character codes. Because most of the methods of

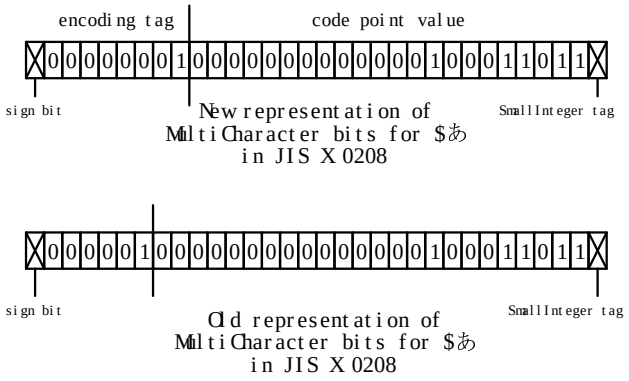


Figure 1. The bit representations of character あ of JIS X 0208 in old and new format of MultiCharacter**. The encoding tag bit boundary has changed, but to keep old instances usable, encoding tag = 4 is reserved for JIS X 0208.**

Character are compatible with this new subclass, we only needed to override about ten methods in MultiCharacter.

How do we use this 30 bit data? In the current m17n Squeak implementation, 8 bits are used for the "encoding tag", which is often refered to as the "leading char", and remaining 22 bits represent the code point in the language/script.

Basically, the 22 bit part is identical to the Unicode code point and the encoding tag is based on the Unicode script definition. Namely, characters in a script defined in Unicode has its own encoding tag value. However, there are exceptions. A character in the unified han area can have different encoding tags to denote its "source standard".

The encoding tag is primaly used for switching the various fonts and implementations that depend on the script. We describe this further in section 6.

Another exception is introduced to maintain the compatibility with the previous m17n implementation. In the previous implementation, the 30 bits were devided into a 6 bit encoding tag and a 24 bit code point. Also, CJK characters are encoded as the domestic standard such as GB 2312, JIS X 0208, and KS X 1001. To make it possible to use the existing old instances in those representation, the encoding tags for those old encodings retain the same bit pattern even though the boundary between the code point and encoding tags has changed.

Table 1 summerize the current encoding tag allocation. In the table, "(U)" after the name represents the script based on Unicode. Non-Unicode encodings appears twice to make it possible to use existing non-Unicode instances created before the boundary was shifted. Before this boundary change, there were 4 encodings are defined, so the new Unicode based encoding starts after 16.

As written in section 2, the definition of Character was changed to Latin-1 based encoding. While most of the characters in Latin-1 were in MacRoman and glyphs were already available, there are several characters and symbols missing from MacRoman. The glyphs for those characters and symbols were bit editted and the existing StrikeFont are modified to have such glyphs.

The encoding tag for a Latin-1 character as MultiCharacter is zero. This means that comparing two characters, no matter they are Character or MultiCharacter, can be done simply by comparing the value instance variables. Unicode standard defines the equality conformance for composited characters. While it can be done at the higher level, we don't provide the full composited character equality in basic MultiCharacter.

4 String and Symbol Representation

Similar to the Characters in orginal version of Squeak, a string of characters, represented as an instance of String class, is essentially an array of 8-bit values. We have added new class called MultiString that is a variable word class.

We change the class hierachy of String to avoid redundant method duplication. While most of the methods in String should be compatible with an instance of MultiString, we can't simply subclass MultiString from String because a variable word class cannot be a subclass of a variable bytes class. We have inserted an abstract class called AbstractString above String and moved most of the String methods to the class. In m17n Squeak image, both String and MultiString are the subclasses of AbstractString and they have specialized methods that depend on the actual character size.

To make these changes, We modified SystemTracer2 [7] to produce an image with modified String class hierachy. Normally, such class hierarchy change can be simply done by editing the class definitions in a browser. However, this doesn't work for String because the virtual machine (*VM*) holds a pointer to the String class object.

To avoid the size change of the subclass array in ArrayedCollection, we first added an empty class called AbstractString under ArrayedCollection and added another placeholder class called DummyString under AbstractString. In the "post-process" phase of modified SystemTracer2, the oop to for DummyString in the subclass array of AbstractString and the oop to the String in the subclass array of ArrayedCollection are swapped.

Thanks to the late-bound and generic nature of Squeak and Squeak VM, the modified image runs on the unmodified VM. After using the system tracer to change the hierarchy,

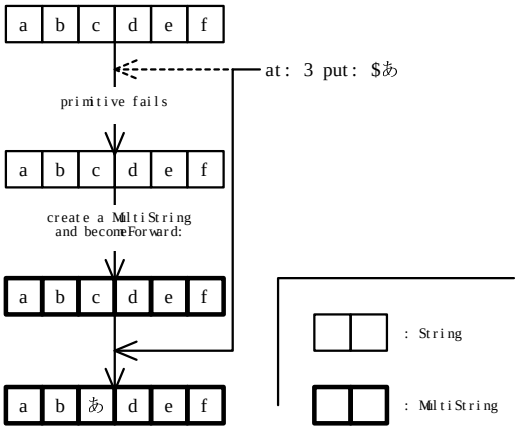


Figure 2. #at:put: **is being sent to a** String **with a** MultiCharacter **as the argument. The #at:put: primitive fails because of type mismatch. The backup code creates an equivalent copy of** MultiString **and then becomeForward itself to the newly created instance.**

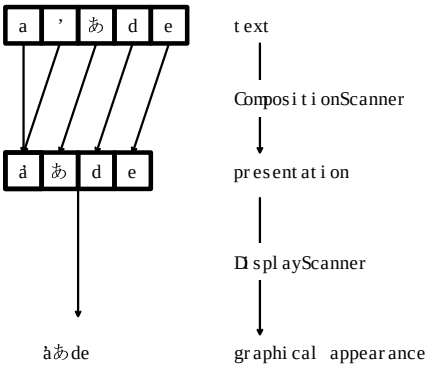


Figure 3. The presentation text for an internal text is created by a CompositionScanner**. Then the result, not the original text, is displayed by** DisplayScanner**.**

moving the methods and the class variables from String to AbstractString, could be done in the live image.

We also added a subclass of MultiString called MultiSymbol which is essentially a copy of Symbol. This class holds symbol tables similar to the ones in Symbol. Again, MultiSymbol implements the same protocol as Symbol and since the VM only uses the oop of a symbol as the lookup key in MethodDictionaries, an image with MultiSymbol class names and method namesruns on the unmodified VM.

5 Implicit Conversion

In the m17n image, there are two classses for kinds of characters and two concrete classes for strings. How are they used?

Similar to the implicit conversions between SmallIntegers and the large integers, Character and MultiCharacter, String and MultiString are chosen appropriately. Namely, if a created instance is representable as an instance of "smaller" class, usually the instance of that class is created.

For MultiCharacters, we potentially create multiple instances that share the same code point. The original Character uses the "flyweight pattern" [8] which create 256 immutable instances beforehand and all subsequent "instance creation" method returns the pointer to those instances. However, for MultiCharacters, pre-creating all instances would not use memory efficiently. This sounds as though the system may end up with another space problem by creating unnecessary MultiCharacter instances. However, in practice, there are not many MultiCharacter instances in the system at any given time.

The conversion between String and MultiString is not as easy as for characters. When you want to put a MultiCharacter into a String, the String instance first becomeForward:s itself into a newly created instance of MultiString whose elements are equal to the string, and then the value of MultiCharacter is stored into the MultiString instance. See the figure 2.

Down converting to String from MultiString can't be "optimal." Usually, a MultiString will remain a MultiString indefinitely. However, when the Compiler parses a string literal in a MultiString and finds that it can be representable as String, it creates a String instance. Thus, a String literals can be created in code that was edited as a MultiString.

6 Text Composition

The basic concept of text composition stays the same for other scripts. A loop traverses an instance of String or MultiString, decides where to break lines and decides where each graphical representation of character is to be placed.

However, the original Squeak's text composition routines must be extended for multilingualized system. In many scripts, more than one conceptual character consistiutes one graphical representation.

For this problem, we separate the conceptual text and its composited result called "*presentation*". Namely, a subclass of NewParagraph, MultiNewParagraph adds instance variables that represent the "lines" of the presentation. In the original Squeak, the task of CompositionScanner is to decide the line breaks for a text and line width

and stores the result into lines instance variable. In m17n Squeak, the scanner creates another Text and sets up line breaks for this Text. Figure 3 depicts the simple example of combining a character "a" and an apostrophe (accute accent) character.

To represent the presentation text, we use the Unicode presentation character code point. This approach simplifies the handling of complex composition but only accepts combinations that have defined code points in Unicode. In the future, a more powerful rendering engine should be present to allow arbitrary glyph combinations.

Many scripts and languages have very distinct text composition and line break rules. Hebrew, Arabic and certain other languages are written from right to left. Japanese users often want to customize the line ending ("*kinsoku*") rules, etc. Because it is hard to write one universal text scanner for all possible scripts, we have implemented separate methods for MultiCharacterScanner and switch them according to the language tag bits of the character. While a sequence of scanned characters shares the same encoding tag, the inner loop of the same scanning method keeps scanning the text. When the loop encounters a different encoding tag, it returns as if an imaginary stop condition was met. Subsequently, other scanner methods are called appropriately.

7 Fonts

The StrikeFont class is used for the glyphs of the characters in a group of characters with a few modifications. In the following, we describe these modifications.

Firstly, there are special "xTable" objects for the ISO 2022 multi octet character sets and Unicode based character sets. The ISO 2022 multi octet characters, namely GB 2312, JIS X 0208, KS X 1001 have only "fixed width" characters so we can avoid storing an array of character offsets; these are calculable by a simple expression.

We have implemented a simple class called XTableForFixedFont. This class mimics an array by implementing at:. It returns the calculated x position in the glyph Form. The glyphs bitmap in the StrikeFont can be relatively large for those fonts, but the mechanism simply works.

Unicode character set are split into groups based on the scripts. In the other words, the characters that share the same encoding tag are treated as if a distinct font. We refer to this split font as a "Unicode-based font". For a Unicode-based font, we attach another object of class XTableForUnicodeFont. The character codes in a Unicode based font may start at a, possibly large, non-zero value. To compact the xTable indexing, the xTable object subtracts the "base offset" from the start value and returns the x table value in the array.

In general, multiple encoding tags may be used in a MultiString, so the fonts for different encoding tags are switched transparently while scanning the string. To implement this mechanism, there is a class called StrikeFontSet added. This class implements all protocols that StrikeFont does but actually hold an array of StrikeFont that represent a same "family" of font for different scripts. When a message is sent to a StrikeFontSet with a string and/or a character as arguments, the appropriate StrikeFont in it is selected based on the leading char of the character in question and the message is delegated to the StrikeFont.

One of the useful by-product of this project is the TextStyle that can now handle TrueType fonts, via what we call TrueTypeTextStyle. By taking advantage of an existing TrueType rendering feature, TrueTypeTextStyle renders the bezier data extracted from TrueType fonts, stores the rendered bitmap in a cache and lays the bitmap out on a Form. For best anti-aliasing, the cached bitmap is stored as a 32bpp Form and rendered onto the destination form by combination rule 34 (pre-scaled alpha-blending). When the scanner sees the TextColor change, the cache is flushed. The cache is also flushed at a full garbage collect occurs.

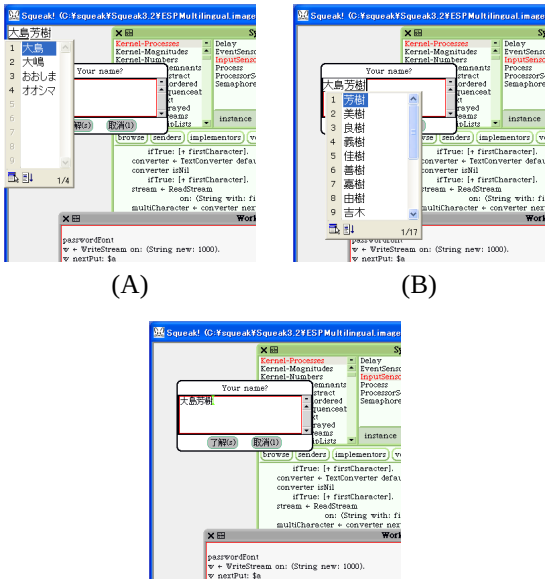
There is a subclass of AbstractFont called TTCFont and MultiTTCFont that represents this TrueType based font. The difference of TTCFont and MultiTTCFont is in the cache algorithm they use. For a "large" character set, which has more than 256 characters, a MultiTTCFont that employs simple LRU based cache algorithm is used. For a small character set, a TTCFont that employs fixed table of bitmap, is used. Similar to StrikeFontSet, there is a TTCFontSet to implement the font switching mechanism.

8 Keyboard Input

To input a script that uses thousands of characters, there must be a process that composes the input from a keyboard, which has less than 100 keys into an appropriate character in the extended character set. The software component that takes responsibility of this composition is often called an "Input Method" or a "Front End Processor". Ideally, this input method would be written in Squeak, but today we rely on the underlying OS.

Some methods in InputSensor are modified so that the keyboard input data from the VM is converted to the appropriate Character or MultiCharacter instance. The default behavior of InputSensor and EventSensor is to pass the each octet to the HandMorph as a character, but for multi-octet characters, the sensor has to combine more than one octets and create a multi-octet character. The correct translation varies depending on the default encoding scheme of the underlying OS.

For the Japanese version of Windows and Mac, the encoding scheme is called Shift-JIS. a Shift-JIS character consists of two octets and the first octet's 8-th bit denotes that



(A) (B)

(C)

Figure 4. Placement of Composition Window.

it and the subsequent octet are the part of the same character. When the keyboard method sees an octet with 8th bit set passed from the VM, it "peeks" at the next octet. If the OS passed a Shift-JIS character, this peek attempt always succeeds so that the method can combine those two octets.

The current implementation of InputSensor treats the Shift-JIS encoding specially, but we are planning to add a generic way to switch the encoding.

The other convenient function on a Japanese system is so-called "inline composition." Typically, the input method shows its own separate window to display the characters being input. To indicate where the string being input goes, the composition window should be placed at the exact position where the string will eventually appear. This sensible composition window placement is called "inline composition". The figure 4 shows the screenshot where the inline composition is not enabled (A) and enabled (B). In (A), the compositing window is placed at 0@0 regardless of the position of the FillInTheBlank window where the user wants to input the text. When the user confirms the intermediate string, the result is passed to the VM and eventually shows up in the FillInTheBlank (C).

The more desirable behavior is shown in figure (B). The compositing window is already shown over the FillInTheBlank and once the user confirm the intermediate string, the string is already in its final position resulting in less surprise and eye-movement.

We have implemented a Windows .dll (plugin) to enable inline composition. To tell the keyboard focus position to the OS, HandMorph>>newKeyboardFocus: is modified and the morph that gets the keyboard focus tells the "composition manager" where to place the composition window. Then the composition manager calls a primitive in the plugin and the primitive eventually calls the `ImmSetCompositionWindow()` Windows API.

9 Reading and Writing Various File Format

The m17n Squeak supports a number of text file formats. The MultiByteFileStream extends the feature of StandardFileStream, and has capability to switch the file format it reads and writes.

MultiByteFileStream has a instance variable called "converter." The family of methods for writing characters and strings eventually calls nextPut: of MultiByteFileStream and that method delegates the request to the converter's nextPut:toStream: method. Similarly, the kernel of reading characters is the next method and this method delegates the request to the converter's nextFromStream: method.

Conceptually, all the converter object has to implement is those two methods. However, for proper peek implementation, the stream has to be able to push back "a character" regardless the its actual size in the file. currentCharSize of the converter returns the octet size for the last character read.

Currently, X Compound Text, GB2312, EUC-jp, EUC-kr, Shift-JIS, UTF-8 are supported.

10 SqueakToys

The most important application written in Squeak to "internationalize" is the SqueakToys system. The SqueakToys already have a mechanism for switching the language in the tiles. A language representable in MacRoman or Latin-1 character set can be used for the tiles in the SqueakToys, once the mapping from the default English keyword to the language is supplied.

Once the multilingualization, or handling the extended character sets, is done, this is also true for the languages that require more than MacRoman or Latin-1 characters. After we have added the capability to handle the multi octet characters, to make the basics of the SqueakToys work in Japanese was not a hard task.

However, there are a lot of details to be implemented. Firstly, the "template" for adding new languages, EToyVocabulary>>templateForLanguageTranslation method, doesn't contain all necessary mapping for strings used in the application. The template method tends to be be left out from the update for the tile system. Secondly, for kids who don't understand English, the strings other than

on tiles must be translated. Such strings include buttons in the navigator bar and the paint tool and the menu items and sub-items in the red halo of Morphs. We ended up with translating about 900 entries to make the Japanese version of SqueakToys ready for Japanese kids.

To enable the translation for the menu items that were not originally the scope of SqueakToys vocabulary translation mechanism, we modified certain call sites for menu creation, StringMorph creation, and alike. At a such call site, the translation mapping dictionary is searched by the original English word as the key and the resulting value is passed to the menu or StringMorph creation method.

The other possible approach to translate the menu is the "callee" side translation approach: namely, the menu creation method translates the passed arguments internally depending on a setting or something. We will consider the upside and downside of those approaches.

11 Conclusion

We have presented the design and implementation of multilingualization ("m17n") effort of Squeak programming system.

The newly added character and string representation are used to hold the extended characters. An object in this new representation is implicitly converted from/to the 8-bit character and string one if possible. The character set for the default 8-bit characters were changed to the Latin-1 and the codes for the extended character sets roughly follows the Unicode definition with encoding tags attached.

The keyboard input is interpreted by the InputSensor. If the multiple octets need to be combined to make a character, the sensor generates the combined multi-octet character accordingly.

Extended text composition routine handles the different composition layout rules. The encoding tag for a character is used to switch the actual implementation of the scanner method to be called. To handle a case where a text contains a sequence of characters that represents a single visual representation, a separated visual presentation text is created.

The glyphs for the entire code space are broken down into separated fonts. Such a font covers a script in the Unicode definition. A font is represented as a StrikeFont object, and the set of fonts that share the same height and family are grouped as an instance of a class called StrikeFontSet. Again, the actual font in a StrikeFontSet is selected based on the encoding tag of a character.

The SqueakToys system is now capable to handle the extended character sets. Because the original language switch mechanism doesn't provide all necessary translation for the end users. We modified the menu and string morph creation sites so that the arguments for them are translated and provide about 900 word mapping rules.

The resulting software has been used by many users. We believe that we are on the right track toward the our goal, which is to provide the collaborative environment for everyone over the network.

12 Acknowledgment

The authors want to thank the Squeak Central team for making the Squeak available public. The open nature of the language was the key to implement this multilingual environment. The subscribers of Squeak mailing list and Japanese Smalltalker's salon mailing list have been giving valuable suggestions and comments. Finally, the experience with the members of ALAN-K Project was crucial to make the project more practical.

References

[1] A. Kay and A. Goldberg, "Personal Dynamic Media," *IEEE Computer*, vol. 10, no. 3, pp. 31–41, 3 1977.

[2] The Unicode Consortium, *The Unicode Standard, Version 3.0*. Reading, MA, USA: Addison-Wesley, 2000, includes CD-ROM. The principal authors and editors of *The Unicode Standard, Version 3.0* are Joan Aliprand, Julie Allen, Joe Becker, Mark Davis, Michael Everson, Asmus Freytag, John Jenkins, Mike Ksar, Rick McGowan, Lisa Moore, Michel Suignard, and Ken Whistler. [Online]. Available: http://www.unicode.org/unicode/standard/versions/

[3] ——, "The unicode standard, version 3.2," http://www.unicode.org/reports/tr28/.

[4] K. Handa and A. Tanaka, "personal communication."

[5] X Consortium Standard, "Compound text encoding," http://www.x-docs.org/CTEXT/ctext.pdf.

[6] J. Murai, M. Crispin, and E. van der Poel, "RFC 1468: Japanese character encoding for Internet messages," June 1993, status: INFORMATIONAL. [Online]. Available: ftp://ftp.internic.net/rfc/rfc1468.txt, ftp://ftp.math.utah.edu/pub/rfc/rfc1468.txt

[7] Anthony Hannan, "Systemtracer2," http://spair.swiki.net/34.

[8] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Massachusetts: Addison Wesley, 1995.