Best Practices for Game Localization

Initial Draft by Richard Honeywood, Vice-Chair
Game Localization Special Interest Group (SIG)
International Game Developers Association (IGDA)
February 1st, 2011

Second Draft by Jon Fung
February 1st, 2012

This is the second draft of a "Best Practices" or "How To" guide for the translation and culturalization of video game content. It is a compilation of suggestions by people who have had years of experience in the field, all members of the IGDA Game Localization SIG. The aim is to help new-comers learn the trade, as well as to offer insights into tricks and tips that even more experienced localization staff can adapt and apply to their future projects.

CONTENTS

CULTURALIZATION

[Note: Contributed by Kate Edwards; these points are excerpted from her handbook on game culturalization, due in late 2012]

What is game "culturalization"?

Culturalization takes a step beyond localization, making a more fundamental examination of a game's assumptions and choices, and then assesses the viability of those creative choices in both the global, multicultural marketplace as well as in specific locales. While localization assists gamers with simply comprehending the game's content through translation, culturalization allows gamers to engage with the game's content at a potentially more meaningful level. Or conversely, culturalization ensures that gamers will not be disengaged by a piece of content that is considered incongruent or even offensive in the game's environment.

Cultural mistakes often prove to be costly for game developers and publishers – not just the loss of potential revenue but the greater effects of negative public relations, damage to corporate image, and strained relations with the local government. In the worst-case, a local government may not only ban the game but take more direct action against the company, including detainment of local personnel for questioning and even incarceration.

Levels of game culturalization

The need for game localization is a well-known necessity within the game industry; however the need for culturalization remains relatively unrealized. Culturalization isn't just a specific task; it's also a broader intent for all international adaptation of content. In its most basic form, content culturalization can be viewed as the following three phases:

1. **Reactive culturalization**: Make the content viable; i.e., avoid disruptive issues to allow a game to remain in the target market.
2. **Localization & Internationalization**: Make the content legible; i.e., perform "typical" localization to allow the game to be understood.
3. **Proactive culturalization**: Make the content meaningful; i.e., adapt and provide locale-specific options to allow the game to be locally relevant.

In regards to these phases of culturalization, some clarification may be helpful:

- Localization is critical but the process of achieving legibility through translation is not the only step required in preparing content for other cultures. This is true for video games as much as it's true for every other type of content.
- It may be argued that a game title should be "legible" before it is "viable." But a government will restrict a game based on sensitive content regardless if it's localized or not.
- These phases are not a hierarchy. As with localization, culturalization takes place in various stages within the typical game development cycle and is a coordination of various tasks and priorities being orchestrated across the entire development process.

Top Four Cultural Variables

The effort of thinking outside our given cultural worldview often makes it difficult for a game designer in one locale to be aware of the issues that could cause problems in another locale. However, by considering at least the following four cultural variables that most often generate conflict between the game's context and local cultures, it is possible to reduce the potential for issues to arise:

1. History: Past and Present

The issue of historical accuracy is one of the most sensitive issues for local markets. Many cultures are extremely protective of their historical legacy and origins, so any alternate or inaccurate history can yield strong, emotional backlash. History is a compelling topic, but it's rarely possible to provide the full context of a historical event in a game. But it's not only distant history that can be problematic but recent history can be a very sensitive topic as the memory of the events and outcome are very fresh in people's minds.

2. Religion and Belief Systems

Game content creators need to be sensitive to the underlying mechanics of the cultures into which their game titles are to be released. In general, a society based on sacred rules tends to be less flexible and yielding to the context in which information appears because they are following what they consider to be a higher standard than human judgment; i.e., if the problematic content appears at all, regardless of context, then there is potential for backlash.

3. Ethnicity and Cultural Friction

Besides the more volatile issues of history and religion, there are many of issues that fit under a broad category that addresses various forms of disagreement, misperception, attitude and ongoing friction between cultural groups. Chief among those is the use of ethnic and/or cultural stereotypes and the perception of inclusion and exclusion with a negative bias towards a specific group.

4. Geopolitical Imaginations

National governments often reinforce their local worldview and the extent of their geographic sovereignty through digital media, including online maps and video games. This involves a situation where the government claims certain territories and they expect those territories to be shown as integrated with their nation, whether it's on a functional map or in the world of a video game (hence the term "geopolitical imagination," as the depiction they're demanding doesn't reflect reality). With some governments, such as China and India, there is no room for error on this issue as they maintain laws that dictate how national maps must appear or how their local political situation must be shown.

See Appendix 2 for examples of games containing these four culturalization variables.

Culturalization Best Practices

The underlying principle of culturalization is that a minor investment of time and effort during the game development process will offset a major loss of time, money and public relations in resolving post-release issues. Fortunately, there are some key steps developers can take to be more proactive about their culturalization strategy.

Gain awareness

1. **Attain a basic awareness**: A key step is to attain a fundamental awareness of the potential for cultural issues; content creators and managers need to understand that cultural issues can occur and in which key markets and which key types of content. For example, most people are aware that China, India, Korea, and the Middle East can be sensitive markets. Also, many people know that certain types of content can become a real flashpoint for backlash, such as maps, flags and historical information.
2. **Ask questions**: The goal isn't to establish subject-matter expert proficiency, but to ask appropriate questions during development. For example, the game *Kakuto Chojin* (2002) contained a brief audio track with a chanted portion of the Islamic Qur'an, resulting in widespread backlash that eventually caused the product to be discontinued. This issue could have been avoided if someone had asked the question: "From where did these lyrics originate and what do they mean?" If something doesn't seem quite right – even if the exact reason isn't known - raise the issue immediately.

3. **Create accountability**: In order for culturalization to be successful, it must be treated as a standard component of the development cycle. This means that responsibility for the process should be assigned to a specific person/team, often times the content coordinators and/or editors. Also, a new bug type "cultural" or "geopolitical" or whatever appropriate should be created in the bug tracking system to ensure the issues are flagged and resolved.

Identify issues

As mentioned previously, culturalization is most effective the earlier it's applied to game content, thus engaging in team discussions around meaning, intent and purpose of characters, plots, environments, objects and so on during the conceptual stages can often catch the majority of potential issues. Here are the fundamentals of identifying potential issues:

1. **Context proximity**: Stated simply, contextual proximity is the concept that the closer a content element approaches the original context in person, place, time and/or form, the greater the potential for cultural sensitivity. Developers should be looking for content that mimics real world locations, buildings, people, events, religions, nationalities, ethnicities and so on, and then evaluating the degree to which the content resembles its real world inspiration.
2. **Leverage external resources**:
    a. Text references: Many reference works can be useful for basic research, such as cultural studies, country-specific guides, symbol dictionaries, encyclopedias of religions and deities, etc.
    b. Online research: Wikipedia, official government websites, non-government organization (NGO) websites, religious organizations, etc.
    c. Local opinions: Accessing the knowledge of people from a specific locale and/or culture can be particularly useful. If you work in a large multinational company, make use of the internal diversity of the company and ask your fellow employees for opinions. Alternatively, you can solicit opinions online in various forums (e.g., Yahoo Answers). This ad hoc opinion gathering may contain subjective viewpoints, but a large enough sample can reveal a clear pattern.
    d. Subject-matter experts: If the above forms of research do not yield clarity, seek out people in different fields such as history, cross-cultural studies or geography.

Assess severity

Just because issues have been identified in the research, it doesn't mean every potential issue needs to be fixed. After identifying potential cultural issues, the key in next stage is to be able to effectively determine the "must fix" issues.

1. **Triage the found issues**: Separate the "overt offenses" – the obvious things that you know for certain will be a problem from the "reasonable risks" – the things that might raise some concerns but won't likely prevent a game from staying in the intended locale.
2. **Document your choices**: Every game publisher has a choice as to whether or not to change sensitive content. Most companies do but there are times when it may not make sense to make even a minor content change because the issue is borderline sensitive. In such cases, it's critical to document the decision-making in a defensive explanation, in case it might be needed if a government or consumers raise the issue.

Implement with precision

Many game designers carry a preconceived notion that culturalization is about making massive changes and rethinking the entire game idea. This is a misperception, and one key reason why many don't confront the geopolitical and cultural aspect at all, as they believe it's going to be too disruptive. This highlights one of the most important principles of culturalization:

1. **Be surgical**: Make the *most minimal change to the least amount of content*. Only change what really must be changed in order to ensure distribution to the game's target market. In the majority of cases with cultural issues, the resolution is a small, precise fix of a specific symbol, or word, or character design; it's usually not a major issue such as the entire game's premise (although this can occur).

Conclusion

Create the game you want to create, but don't forget the global, multicultural audience who will be participating in your vision, and hopefully enjoying it without any cultural disruption. Well-executed culturalization within a development cycle isn't turnkey; it takes time to implement successfully. However, the benefits to a company's content quality, government relations, and public image amongst local gamers will prove to be a valuable long-term investment.

INTERNATIONALIZATION

Internationalization, abbreviated as I18N, is the process that enables game localization to take place. Before internationalization a product can only display game content in one language. After internationalization, a products code base, architecture, and user interface are capable of processing and displaying game content in multiple languages. An internationalized product does not contain any elements that differ by locale.

User Interface

Fonts

Asian languages, like Japanese, Korean and Chinese, predominately use fixed-width fonts. If the original Asian-version's programmers are not willing to reprogram their system to handle proportional fonts for European language versions, then the European language versions will suffer from wasted space between letters, often causing the look of the game to suffer, as well as causing the translation's quality to drop due to reduced line width limits.

While this should no longer be an issue on most PC and console games, it still seems to be an issue in hand-held games.

To further maximize the space on screen, apart from a proportional font, it may be good to utilize a font display system that allows for kerning (different spacing between letter combinations so that they optimize the white-space underneath overhanging letters etc.).

Besides being a headache for programmers accustomed to fixed-font languages, the downside is that it makes line width limit checking more complicated for the translators. It becomes necessary to create simple line-width checking tools or Excel macros that add up the pixel width of each letter and can tell you whether it is within the space limits set aside for that string on screen. But the payoff in translation quality and improved look of the game is definitely worth the effort.

It is important that the translation and development team work together early in the development cycle to select the fonts. The translators need to be able to verify that all the letters used in their language are available in the selected font. Further, you need to decide on the font size and font type, to make sure it is readable when displayed on screen, as well as to determine the pixel widths of each letter for use in width limit testing (i.e to make sure the translation will fit in each allocated space on screen). Only a native speaker of a language can tell you whether the "feel" of the font is right, from whether accents marks look right in European languages, or whether the Chinese characters are readable in Asian languages, so always make sure you do a font approval step before translation begins.

U.I. and Message Windows

Once you start to deal with multiple languages, you will find the layout of the text once displayed on screen game will inevitably start to need some tweaking between languages. In order to better assist this, we suggest:

- That you allow "speech bubbles" to auto-resize for message windows or display areas that are more flexible. (The one-off cost to program such a system will save your dev team a lot of time resizing things by hand later.) If a fully automated system is too much, then at least allow the sizes to be resized by specifying the x,y,w,h coordinates in the text files...then that way the translation team can fix them in each language instead of overwhelming the development team with such requests.
    - Alternatively, if all text boxes must be identical across all languages, size them for the length of anticipated translations opposed to the length of English text. The rule of thumb is to approximate translations as being 30-40% longer than English.

- That you implement an automatic text wrap around system. i.e. instead of asking the translators to hardcode each line-break in by hand (then having to reshuffle the text around constantly to make it look like it is broken at the correct place on screen), have the game system do so automatically. (Also allow for an over-ride for times when you don't want words automatically broken across lines...but more on that latter).
- This is practically a must for PC games that allow for resolution and screen sizes to be changed by the user!
- (As another option when appropriate...) That you implement a font expansion-contraction system so text perfectly fits the allocated area. i.e. in the menu headings etc, if the text slightly goes over the allocated space, that the system shrinks the font to fit. (Better than cutting off translations mid-word.)
- (Alternatively, when appropriate...) Allow for text to scroll or expand out into a pop-up screen. That way if space is limited in the GUI, the player can still see the full text when highlighted, instead of having to decipher abbreviations or cut-off words everywhere.

Subtitles in Pre-rendered Movies

To save disk space, as well as to save your development team from having to re-render multiple versions of the same movie, try to implement a system where the subtitles are displayed over the top of the movie real-time when played, instead of "burning" them into the movie files. If creating a system is a lot of work, just think of the cost every time a bug-fix has to be made in those subtitles for each language. (And it is very rare that the subtitles will look right and be timed perfectly first time, so a more flexible system than re-rendering everything over-and-over is highly recommended.)

Best practices for subtitles include:

- Incorporate a san serif font with a bright color and dark outline so it is easily visible against both light and dark backgrounds. (see Red Dead Redemption for an example)
- Position subtitles in the same area of the screen throughout the game; preferably the bottom third of the screen except when it interferes with the interface.
- Display each set of text for 3 seconds, left aligned, allowing 96 characters per line, and up to 3 lines of text at a time. Any more lines at once become difficult for the user to read.
- Use static text, avoid scrolling text whenever possible.
- When two people are having a discussion, indicate changes of speaker by adding a hyphen (-) to the start of their line, especially if the new speaker is off screen. (This should also happen in English subtitles.)

Architecture

Localization assets should be stored in a content management system with a folder structure that is accessible to the localization team. Some suggestions which are commonly used to ease exporting / importing is to store the assets by level, game location, or by language.

Language and Country Codes

While it might be tempting to just add single letters (such as "E,F,I,G,S,J,K, or C") to represent languages or regional versions the end of file names, directory names, or program labels, you will eventually come across issues such as US English versus British English, European Spanish and Portuguese versus Latin American Spanish and Brazilian Portuguese, and even mainland Simplified Mandarin versus Taiwanese Traditional Mandarin versus Hong Kong's Traditional Cantonese. Further, there can be confusion over which code represents which language (e.g. whether German should be written "Ge" or "De" for Deutsh). To solve confusion over languages and country codes, a good idea is to stick to the ISO standards that are recognized across industries...

http://en.wikipedia.org/wiki/List_of_ISO_639-1_codes
http://en.wikipedia.org/wiki/ISO_3166-1
(or http://www.loc.gov/standards/iso639-2/php/code_list.php)

For example, if you combine a two-letter language code with a two-letter country code, you have short but readily recognizable ways to differentiate language and regional versions. (When one language is used in multiple countries, such as Latin America, you can just pick the most representative country, such as Mexico.) Either use a dash between the country and language code (PT-BR), an underscore (PT_BR), or even more concisely if your system allows it, write one lowercase and the other uppercase (ptBR)

This gives us:

enUS	American English
enGB	British English
deDE	German
esES	Castilian Spanish
esMX	Latin American Spanish
frFR	(European) French
itIT	Italian
jaJP	Japanese
koKR	Korean
ptBR	Brazilian Portuguese
ptPT	Iberian Portuguese
plPL	Polish
ruRU	Russian
zhCN	Simplified Chinese (Mandarin)
zhTW	Traditional Chinese (Mandarin)

It may take some time to become accustomed to using these, but it will clear things up immensely.

Localization Tools and File Formatting

Just as development teams use a lot of different tools to manage their resources (Visual Source Safe, AlienBrain, etc.) and make programming smoother (Visual Programming Environment etc.), translation teams will likely need proper tools to help their work go smoothly. While a lot of work will likely be in plain text files or Excel files, File Management tools, Translation Memory tools, electronic dictionaries, and file comparison tools (BeyondCompare or even old “Diff”-like tools) will surely optimize the translation process, especially when working in changing resource files or in large teams. We will not go into details of such tools here, but we encourage you to find the best solution for your particular development and translation teams’ needs.

While on the topic of file formats, if you are using plain text files, be aware of file format differences particularly once you come across Asian languages. Rather than dealing with corruption issues between Extended ASCII and JIS encoding etc, it is probably wiser to stick to an encoding that allows all of your target languages to co-exist from the outset, such as UTF-8 or similar.

A common pitfall in games is the issue of text used in graphics. Whether 2D textures or 3D polygon models, the dev team needs to track where all “text” appears in their game in order for it to be localized correctly. Be prepared to have to redraw all graphical text for each language version (or find a work-around that allows you to use graphical icons and symbols instead of text to reduce the workload on graphic artists). These need to be provided to the translators and have their changes tracked just like all other text resources.

Resource Files and Character Encoding

It is best not to hard program UI text into the source code (or for that matter any game script that could be problematic if accidentally changed by the translators). And it is imperative that files are formatted logically. If "events" or storylines in the game are broken across files or are not in any order, please make sure there are comments or some labeling system in place that will help translators understand the order of the text and where it will appear in the game.

Indicate which platform the strings belong to; particularly for saving, loading, or other functions affecting platform compliance. The localization team will need to put special care in translating these strings to use the terms set out by the console manufacturers.

Obviously things will be cleaner/smoother if all files are in a similar format to each other. Particularly if either the development team or translation team are using convertor tools, the fewer formatting changes, the less work (and chance of error) that goes into converting and re-converting files.

Depending on the file encoding too, you might run into memory issues. If Asian languages use double-byte character sets, and also expect English and European languages to do so, they will soon see that the lengths of the strings might take up more memory than expected. (English uses on average 1.5 more characters than Japanese, say. Spanish and German may use more than double the amount of letters than Japanese.) To avoid memory overflows, or simply wasted memory, use an encoding that keeps European letters in single-bytes (such as UTF-8 or other variable byte-length encoding).

On the flipside, for Chinese, Korean, and Japanese, if Western developers have trouble fitting the entire Asian font set into memory (say on handheld games) then a trick is to just load the actual characters or glyphs that appear in that part of the game into memory real-time. Otherwise, if you need to load the entire font, you can talk with the translators about what segments of the font they may not need and at least remove those from the font file. For example with Japanese, most phrases can be translated using a set of 2,000 characters (the number of characters taught to children in school).

Programming

General Programming Tips

It goes without saying that you should avoid "hard coding" text within the program, as it means a programmer will have to fix those strings by hand every time, instead of just letting the system do it automatically at build time. Furthermore, when text is hardcoded, the programmer will have to separate the text to be translated from the program source and re-input it by hand...or otherwise hand over the source code to translators to find the translation in (and risk them accidentally changing programming code by mistake as well). The best practice is to isolate all text strings used in game into a resource file for each language.

As mentioned in the UI and Message Windows section, when you want to center text or position it in set places, rather than hard coding the coordinates, it is better if you allow the system to center the text or position the text for you based on the length of the target strings. (Otherwise, once again, a programmer will have to correct things by hand each time. An alternative for when text needs to be positioned or centered but the system cannot automatically support it is to allow the coordinates to appear in text files or separate resource files, so that the translators or QA can make the positioning changes for each language between builds instead of taking away precious programmer time.)

Resource files should be separated by language. While it might look neat to put all languages in separate pages or columns in an Excel file, in practice it means the different language translators can't all work on the same file at the same time and someone will end up having to copy & paste all the language's texts back into one file, opening the process up to human error and wasted time. It is recommended that each language's files exist in their own directory and are copied from there at build time.

Also to assist the localization team, include a list of the source language text changes between all build versions. This will minimize the chance of the localization team missing a new or edited string.

Using XML for Translation Text

Some companies are finding the use of XML (Extensible Markup Language) a great alternative over plain text files or Excel tables to do their project's translation in. For instance, if you know that some languages only need one version of a string but others need up to six versions of the same string (say because the latter has singular/plural or masculine/feminine/neutral in their grammar while the former does not) then you can use XML tags to allow for both cases. So instead of making all languages translate 6 versions of the same string identically in their language, they could just translate it once. While the languages that need more versions can add them in where needed. (In your program's memory you might allow for the maximum number of combinations for each line, even if most end up as NULL strings in the end. And as the XML file is parsed and compiled, the actual strings that exist in the XML are placed into the source code and the system is flagged to branch to those strings runtime.)

That way, Japanese, Chinese, and English can merely have shopkeepers greet you with the equivalent of a simple "Hello!" but in Italian you would have the equivalent of "Hello Sir!", "Hello Madam!", "Hello gentlemen!" and "Hello ladies!" each with appropriately defined XML tags that make the game system record that singular/plural and masculine/feminine branches exist and branch real-time according to context inside the game.

XLIFF (XML Localization Interchange File Format)

While many platforms, game engines, and other technologies store text resources in various file formats, supporting these formats can be troublesome to localization teams. Primarily it complicates the cycle of requesting and implementing translations as each format requires a separate workflow. It also hinders your team's ability to develop quality and cost saving translation tools.

In 2002, the XLIFF format was established to solve this problem and provide a standard format for all developers to exchange text translations with localization teams. Implementing this standard requires processing text resources from its native format into the XLIFF schema for translation requests. Translation implementation requires post-processing from XLIFF back to the native text resource file format.

For further information on XLIFF, and supporting tools, please reference these links:
http://docs.oasis-open.org/xliff/xliff-core/xliff-core.html
http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=xliff
http://en.wikipedia.org/wiki/XLIFF

Order of Variables in Text

When using C or C++'s printf or similar text output functions, remember that the order of %s and %d variables into text strings will likely change across languages. For example:

msg="**%s** bought **%d** **%s** for **%d** dollars.";

will probably change order in Japanese and Korean to...

msg="**%s**が**%s**を**%d**個買って、**%d**ドルを支払いました。";

(i.e. the order: character_name, number, item_name, cost
becomes: character_name, item_name, number, cost.)

If you do not create a system that allows for the translators to freely change the variables inside the target text, then the programmers might have to change the order of each string's variables to match the order in the target languages.

Note: as we have been speaking about singular/plural tokens above, we can show how the implementation of the above system can make the resulting text much more natural and correct.

e.g. "John bought 6 potion for 1 dollars."

...is obviously wrong, but if we implement the token system so the text looks like the following...

msg="<Cap><Character1> bought <IF_SING_Number1><INDEF_ART_SING_ITEM_2><ELSE_NOT_SINGLE_Number1><Number 1> <PLR_ITEM_2><INDEF_SING_ITEM_2><END_IF_SING_Number1> for <Number2> <IF_SING_Number2>dollar<ELSE_NOT_SING_Number2>dollars<END_IF_SING_Number2>.";

...then, if implemented correctly, the above system can result in all of these potential outputs...

John bought an apple for 1 dollar.
John bought 2 apples for 2 dollars.
Mary bought a pear for 3 dollars.

and so on....

Date, Time, Currency and Number Formats

All locales have different formats for displaying dates and figures. Syncing up with your localization team prior to game production will enable development teams to establish all affected areas of the game and which format to implement for each language.

Here are some standard formats for the most common languages.

	US English	*French*	*German*	*Spanish*	*Italian*	*Japanese*
Date	mm/dd/yyyy	dd-mm-yyyy	yyyy-mm-dd	dd/mm/yyyy	dd/mm/yyyy	yyyy年mm月dd日
Time	hh:mm:ss am/pm (12-hour clock)	hh:mm:ss (24-hour clock)	hh:mm:ss (24-hour clock)	hh:mm:ss (24-hour clock)	hh:mm:ss (24-hour clock)	hh:mm:ss (24-hour clock)
Decimal Separator	period (.)	comma (,)	comma (,)	comma (,)	comma (,)	period (.)
Thousand Separator	comma (,)	space ( )	space ( ) or period (.)	space ( )	space ( ) or period (.)	comma (,)
Number Example	15,631.87	15 631,87	15 631,87 or 15.631,87	15 631,87	15 631,87 or 15.631,87	15,631.87
Currency	$12,345.67	12 345,67 €	12 345,67 €	12 345,67 €	€ 12.345,67	¥ 12,345
Ordinals	1st 2nd 3rd ...	1er 2e 3e ... or 1re 2e 3e ...	1. 2. 3. ...	1º 2º 3º ... or 1ª 2ª 3ª ...	1º 2º 3º ... or 1ª 2ª 3ª ...	1目 2目 3目 ...

One of the best ways to implement these various formats is to include a string in the resource files for each number format type that are translated. This way it will be up to the translation team to correctly identify the order of year, month, day as well as the different characters used to separate numbers.

When currency is localized in game, as well as paying attention to unique currency symbols for each language, you should also approximate exchange rates. For example, in the United States a game selling a cheeseburger for $2.00 is okay. However it would be unrealistic to display the price as ¥ 2 in the Japanese version; it would more likely be ¥160 (if you assume an exchange rate of 80 yen per US dollar).

Automatic Line Breaking and Wrapping of Text

Implementing an automatic line breaking or word wrapping system in games with a lot of text will greatly reduce the chances of text overflow bugs. It will also greatly speed up both work time and QA time when compared to having the original writers and translators input all line breaks by hand.

Further, when the same text is displayed in several different places with different widths, it will save you having to prepare multiple versions of the same text (or worse, force the translators to prepare the text for the worst case width and leave the rest of the space unused on other screens.)

That said, there will be times when you need to override the automatic line-breaking algorithm. For instance, the console hardware companies often enforce a rule that certain terminology not be broken across lines. In this case a "non-breaking white space" symbol can come in handy. Either: a special hidden character code within the Unicode font set (0xA0); or, a token such as "**<nbsp>**" or "_" (underscore) that is converted run time to look like a blank space but is an exception to the line breaking algorithm. (Further, if your line break algorithm also breaks at hyphens, you may need a special non-breaking hyphen symbol or "**<nbhy>**" so words that cannot be broken at hyphens are not...although this case is probably rare.)

Further, your game text may have special circumstances like this:

"Brigadoon ->
<- Xanadu"

If left to the automatic line breaking system alone, then this road sign text might get displayed like this:

"Brigadoon -> <-
Xanadu"

Once again, a hard line break character or token such as "**<lb>**" will resolve these issues without programmers having to step in and hardcode exceptions each time such issues occur.

Exceptions for Punctuation in English, German, Spanish, and Italian

Your basic automatic line-breaking algorithm will work as follows: if the text goes over the length of the window for this line, then find the last letter that fits on screen and scan the text back from there until you find either a space " " or a hyphen "-". Then move all text that appears after the space or hyphen down to the next line and repeat the process on the next line. In the case that no space or hyphen is found when scanning backwards along the line, then you have no choice but to just forcibly break the text mid-word at the letter that goes over the line limit. (Grammatically speaking when you break a line mid-word, it would be more correct to break at a syllable and add a hyphen at the end of the line, but that is more advanced than most games need, and they can be dealt with as rare exceptions by utilizing hard line breaks to resolve them.) One small optimization you can make to the algorithm is to only scan back 15 or so letters instead of the whole line (particularly if you implement the "Hidden Line-Break Token" system below).

Treat commas",", periods (full stops) ".", exclamation marks "!", question marks "?", semi-colons ";" and colons ":" as if they are part of the previous word they are connected to, so as to void ever having these appear by themselves on the next line.

Also be careful of "em-dash" and "en-dash" (long hyphens like "—" that have the length of an m and n, respectively) as well as the Spanish "raya" (approx. triple-length hyphen used like a quotation mark which has character code 0x97). These can be moved to the next line. However, often when the font or text display system does not support these characters, the translators may use two hyphens as a substitute...so your system may have to make exceptions and break either before the first of two consecutive hyphens or after the second of two consecutive hyphens.

Note: Although German uses 99-shaped quotation mark at the bottom of the line, and Spanish uses an inverted exclamation mark at the beginning of a sentence, these follow the same rules as the other punctuation marks listed above as far as auto-line breaking is concerned.

(German quotes look like this: „ ‚ ‚ ‚ “ and have the code 0x84 and 0x93)

While on the topic of punctuation, it is good to make sure that translators have the option to use both single ‘’ and double “”quotation marks. British English uses single quotes for quoting text, while American English uses double quotes...but both are used alternately when quoting text within another quote. Ficklers for style might also like to use a straight apostrophe instead of reusing the closing single quote mark (i.e. not the 6 or 9-shaped quotes, but a vertically straight apostrophe mark that has the code 0xB4). This helps differentiate single quotes from apostrophes in text that has a lot of quote marks...although not

many companies go to that extent to improve readability.

If the game system or translation tools for whatever reason cannot handle punctuation such as “” "`‘’ then you might consider implementing tokens like <66>,<99>,<11>,<6>,<9>, and <1> that are replaced at either compile or run-time to represent those symbols as a workaround. (These are just simple to visualize what they are and are easy to enter by the translators if necessary.)

Exceptions for Punctuation in French

French has some different punctuation rules that need to be taken into consideration when implementing an automatic line-break system. Exclamation marks “**!**”, question marks “**?**”, semi-colons “**;**” colons “**:**”, and double-quotation marks “**«**” and “**»**” (i.e. 66-shaped and 99-shaped quotes) generally have a space appear before them. To deal with this, we suggest the following implementations:

1. Force the French translators to use a non-breaking space (0xA0), or alternatively a token such as “**<nbsp>**” that is treated as non-white space but is displayed as a space at run-time. Prior to delivery, French translators should search text for instances of punctuation preceded by a breaking space and change them to non-breaking spaces. As a best practice, localization engineers that are implementing localized text should occasionally check French text for missing non-breaking spaces and point them out to the translation team.
2. Make your line-break system deal with French punctuation correctly when displaying text in the French version of the game. That would mean you treat “**Bonjour !**” as one unit of “word + space + special punctuation”. (The preferred alternative.)
3. A cheap and nasty work-around is to add a space’s width of blank pixels in front of the following characters in the French font: “ **!**”, “ **?**”, “ **;**” “ **:**”, “ **«**” and “ **»**”. (This can be done simply by adding a space’s width worth of pixels to the width of these letters in the font width data.) The French translators will need to be told to not add spaces before these symbols when translating, but trust that it will look okay once on screen, without the need to re-program the line break algorithm. However, you may also need to provide tokens (like <!>, <?>, <:>, <;>, <“>, and <”>) or alternative characters that look like normal-width versions of these fonts for special cases such as “What the...!?!” where punctuation does run next to other letters without spacing.

Japanese, Korean, and Chinese Punctuation

Although JKC are not as strict on where you can or cannot break text, they do have some general policies on punctuation that need to be followed...even if you use a simple fixed-width font and line break algorithm. (Note, Asian languages do not use spaces that much, so often you will be splitting the text in the middle of a stream of letters where it happens to hit the end of a line more so than searching for a space to break at.) The exceptions are similar to European languages: periods/full-stops “。”, commas “、” and “，”, closing quotation marks “」” and “』”, and the hyphen-like center dot “・” should not appear at the start of a new line, and the opening quotation marks “「” and “『” should not appear at the end of a line without any text after them. Basically allow some buffer space at the end of the text display area for a period or comma-like character to be squeezed in at the end of a line if it happens to be the character to be broken at. Otherwise you will have to move the preceding character across to the next line with the punctuation mark so that it won’t appear alone at the start of a new line. On the other hand, if you are about to break at an opening quotation mark-like character, then let it move across to the start of the new line with the characters that come after it, in order to maintain correct punctuation rules.

Visit this link for further information:
http://en.wikipedia.org/wiki/Line_breaking_rules_in_East_Asian_language

Advanced Option for English and European Languages: Hidden Line-Break Tokens

Particularly when text is generated by the game system real time, or when the text is displayed in differing (or re-sizeable) display areas, long words can start to throw off the balance of the length of lines when displayed on screen, reducing the look of the text windows or UI. We can take a tip from print media and implement a system where long words are properly broken at the nearest or most-logical syllable, and a hyphen is added at the end of the line indicating the word carries across to the next line.

If you provide the translators (and source language writers) with a "hidden line break token," they can put that token in long or problematic words, in order for automatic line breaking system to use as a guide to break the text at. Instead of a long token like **<hlb>**, if the target text won't be using the tilde "**~**" or caret "**^**" symbols on screen, you could use these as tokens, and insert them between syllables of long words. Your text display system would never actually show these on screen, but your automatic line break routine could also break at these when searching for the nearest hyphen or space within the last 15 letters of the text string.

e.g. Logically if you were handwriting or displaying text in print media, you would break "**broadsword**" at "**broad-**" and carry "**sword**" to the next line. So the translators could write "**broad^sword**" and the system would do exactly that if the word happened to carry across the end of the current line.

Similarly if you had "**supercalifragilisticexpialidocious"** causing your game trouble when wrapping text between lines, we recommend you adding in a few hidden line breaks at the major syllables: "**super^cali^fragilistic^expi^ali^docious**" even if you could go all out by entering them at every syllable with "**su^per^ca^li^fra^gi^lis^tic^ex^pi^a^li^do^cious**". This freedom will improve the look and readability of the text once on screen. (While it might seem a little extravagant in English, for languages with longer words such as German, this can really make a difference to the optimization of screen space.)

However the trade-off is of course the issue that translators will need to add these symbols into their translation, taking up time and causing headaches with auto-correction and spell checker functions, but a good preview or line checker tool will go a long way to improve matters here.

Even more advanced way: An alternative to adding hidden line break tokens directly to the translation text is to prepare a dictionary or look-up table for each language that lists where common long words can be broken. (i.e. a table that tells the system at build time to replace all occurrences of "**broadsword**" with "**broad^sword**" etc.) That way the translators don't have to input these into their text, but instead, when the files are preprocessed for integration into the game system, any words in the dictionary/look-up table will be replaced then and function as if the translators entered these by hand throughout the text. This will be less of a burden on the translators than having them enter these tokens by hand every time throughout translation (not to mention having to deal with spell-checker and auto-correction flagging their text too).

Allowing the Use of Special Symbols

We mentioned the workaround for when punctuation like **“” "`'** cause trouble with either the game system or localization tools by implementing tokens like **<66>**,**<99>**,**<11>**,**<6>**,**<9>**, and **<1>** that are replaced at either compile or run-time to represent those symbols. Similarly, you could control the use of en-dash, em-dash and rayla by using **<->**, **<-->**, and **<--->** as workarounds as well. Here we use <xxx> to symbolize tokens, but they could just as likely be {xxx} or /xxx or any other representation that works in your system.

A further option, if your font has specialized symbols that are not easy to access by typing into the word processor, text editor, or Excel, then you might allow the translators access to those via a token system of some kind. Here are some examples of glyphs found in Asian languages that European language translators may want to have access to also (if they share the same font or these double-byte symbols can be made available to them in their single-byte font as well)...

← **<l_arrow>**, → **<r_arrow>**, ↑ **<u_arrow>**, ↓ **<d_arrow>**, ⇔ **<2way_arrow>**,⇒**<doub_arrow>**, ☆ **<star>**, ★ **<blackstar>**, ※ **<snow>**, ～ **<swungdash>**, ♪**<music>** (or **<.|>**), △**<triangle>**, ▲**<blktriangle>**, ▽**<invtriangle>**, ▼**<invblktriangle>**, □**<square>**, ■**<blksquare>**, ○**<circle>**, ●**<blkcircle>**, ℃**<Celsius>**, ° **<degree>**, ①**<circleone>**, ②**<circletwo>**, ③ **<circlethree>**, ④ **<circlefour>**, ⑤ **<circlefive>**, ⑥**<circlesix>**, ⑦**<circleseven>,** ⑧**<circleeight>**, ⑨ **<circlenine>**, ⑩ **<circleten>**, ♂ **<male>**, ♀ **<female>**, ＋ **<plus>**, ＝ **<equal>**, and ／ **<slash>** etc.

This is an easy to implement extension that offers all languages access to useful or fun symbols in their translations as well.

Grammatical Tokens

To make your text grammatically correct and more naturally looking in your target languages, you might like to consider implementing a system like the following tokens. Pick and choose to match the requirements of your title. For titles with not much text, it might be reasonable to just hard-code branches into the source code and prepare exclusive strings for each case. But in large RPGs or MMORPGs where the combinations of variable text is practically unlimited, token systems are practically a must if you want readability and understandability.

WARNING: SQUARE ENIX has the intellectual property rights pending on some of these implementations (Japanese patent application 2003-178063), so you might want to understand the concepts but find a different implementation that suits your system. For safety's sake we will not list any source code here to prevent accidental copyright infringements.

(That said, SEGA owns the patent for name entry screens and the replacement of the player's name into game text, but almost all games use such systems without fear of a legal suit. So it's up to the developers to decide what implementation to make and judge the associated risks.)

As before, we are using <xxx> to indicate a token, but you could equally use {xxx} or /xxx or whatever works for your system.

- **<Cap>** token to make the first letter of the following variable's text upper case.
- **<CAP>** to make all the letters in the text that follows upper case.
- **<bold>** and **<italic>** tokens for bold and italic font access, if your system supports it.
- Color tokens such as **<red>** or **<white>** if your system supports colored text
- **<wait x>** token to pause the display of the text that follows for a number of frames. Can add an extra level of "performance" to the text by pausing for emphasis, and may help subtitles match the timing of the spoken dialogue.
- Article tokens (like replacing the number 1 with "the, a, an" in English, or even more combinations in French)
- Singular/plural item and mob name tokens. (If you have many names, you will probably need a look-up table system instead of just trying to automatically add "s". Even English has many exceptions such as goose->geese; genius->geniuses; fish->fish. Also make sure to handle exceptions for when you count a pair of objects as one.)
  - Note: Asian languages such as Japanese and Chinese do not use singular/plural very much so it will require those languages' programmers to understand concepts they do not have in their own language.
  - Note: Be careful of 0...in most European languages it is treated as a plural number, but in French it is treated as singular. Make sure your system can handle this exception if you are doing French or other such languages.
- Masculine/Feminine/Neutral grammatical tokens to handle "his/her/its"-like grammar variations in European languages.

- Tokens that branch text depending on whether the first letter is a vowel or not.
  - A simple test of the first letter may work in most cases. However you may need to extend it for French to encompass silent h's, y's and other special cases. For large games, it generally is better to have a lookup table for all names that lists whether a name should be treated as a vowel or consonant instead of teaching code to handle every exception in every language.
- Tokens that branch text depending on whether the last letter of a variable's text is a vowel or consonant. This is particularly useful in Korean, where articles come after the subject/object in a sentence, and need to branch on whether the last letter is a vowel or not.
- Tokens that branch text on whether the last letter of a variable's text ends in an "s" or "S". (Extend this for German to also handle z, Z, x, X, and ß (shaffen-S, the German double-s.) Many languages such as English have rules for possessives where you add apostrophe s ('s) to most nouns, except in the case where it already ends in s and you just add an apostrophe instead.
- Tokens that check whether the text replaced by a variable is a proper noun or not.
- Tokens that handle German cases (nominative / genitive / dative / accusative) and Russian declensions (singular / plural / reflexive x 1st / 2nd / 3rd (Masculine / Feminine / Neutral) x nominative/ genitive / dative / accusative / instrumental / prepositional).
- Tokens that display the correct counters after numbers in Japanese, Korean and Chinese. (i.e. usually you have to put a character after each number to represent what kind of thing it is counting, so small furry animals have one type of counter, bigger animals have another, flat objects have one symbol, small items another... a concept that doesn't really exist in European languages.) A simple flag after each name in the list of names should suffice. Then you just have the token reference that flag to decide which counter to print.

We suggest you talk to your translators about their requirements for each title and see what your development team's abilities, time, and budget allows. While we advocate correct grammar in all languages, we also realize that it becomes a business decision on how correct you want your game to appear in each language.

Note: the trade-off for the added beauty and grammatical correctness of these tokens are: added complexity for the programmer and translators, and added level of QA testing to make sure a more flexible text system handles more combinations of text display properly. But for MMORPGs, RPGs and other text-centric games, these make a huge difference in the understandability and quality of the title. (e.g. once human-to-human interaction comes into play, you want to make sure singular/plural and who is doing what to whom is very clear.)

See Appendix 3 for a detailed explanation of grammatical tokens with actual token names.

PAL vs NTSC TV

One aside to dev teams who are making games for consoles that are expected to run on old-school TVs, be sure to take in regional TV standards into your programming. For example if you develop games in the USA or Japan, then decide to release a European version, you will run into issues with frame rates (60fps vs 50fps) and screen sizes. If you do not program in a way the program runs independently of the TV frame rate, your PAL TV version may experience a 1/5 gameplay slow down, blocky animations and pre-rendered movie playback, and likely have big black bands at the top and bottom of the screen. As the newer console hardware and digital HD TVs are alleviating this problem, we won't go into detail about how to program around it or how to re-render movies etc here at this time. We just bring your attention to it as it can be one pitfall of localization that many development teams hit far too late in the development cycle.

Advice for Development Teams

Work with your Localization Team as early as you can on your project.

1. By taking localization's needs into consideration during development, it will reduce the amount of work you have to repeat over-and-over for each language version down the line.
2. By making your source language resources more systematic, it will reduce the time needed to translate the content and reduce translation mistakes and lime-limit overflows, thereby increasing the quality and reducing the amount of bugs your development team will need to correct when they can be moving on to their next title.
3. By working towards standardized formats and processes, the time by all parties spent on localization will be reduced as everyone involved won't need to re-invent the wheel or spend time adapting to the differences between development teams.
4. Clear communication is essential to allow your product to smoothly pass through your regional distributor's and console makers' checklists, as well as regional ratings boards. (e.g. be sure to provide the translators with updated Interface Guidelines or other documentation from the console makers where necessary, as these may have change, causing you delays to market when you have to remake build and resubmit your masters.)

Remember, localization is your pathway to a wider audience. If you want people in other cultures to enjoy your title as much as the original language's version (and to keep buying your products), then the more information you give your translation team and the more willing you are to make changes if needed, then the better.

LOCALIZATION

Localization, abbreviated as L10N, is the process of creating a product suited to a specific locale. It includes producing localized resources or assets, implementing the assets in an internationalized build, testing, editing and bug fixing. The localization assets can include translated resource files, localized voice over files, translated packaging material, and translated metadata among others.

The localization process of a video game can be broken down into the following stages:

- Familiarization
- Glossary and Style Guide Creation
- Translation
- Voice Over Production
- Linguistic Quality Assurance
- Master Up and Sign Off

Translation, editing, recording and QA may overlap and happen in parallel to optimize scheduling. Refer to the sample schedule in Appendix 4 to help visualize how each stage may overlap.

Familiarization

No matter how small or large the project, translators need to have read or seen the content before they begin translation in order to gain a feel for the subject matter. You expect a novel or movie translator will have read/seen the work before they translate it, so why wouldn't we expect to give context to the translators of a game (that is much more complicated with multiple story branches and text files with no logical order to them). How much time to allocate and to whom depends on the size of the title and budget limitations. A minimum of 3 days of playing the game and reading background documentation is a good starting point for small titles, with another 2 days to familiarize with previous localized titles in the series when appropriate. For large MMORPGs, you may need to allow at least a month to play through all the content (even using debug or cheat commands). If the title is still in production, then providing early builds and design docs may have to suffice until playable versions are available.

Translators should keep a record of names and ideas as they play through the game content. In order to track their pace, it may be wise to ask them for daily or weekly progress reports. If they use debug commands they should also play a little of the game without them to know what elements are important to gameplay. They should also check out all modes and areas of the game, including any multiplayer mode and online components. The more they cover in this early stage, the less they will need to stop translation to refer back to the game later. This list of terms should be checked against the glossary discussed in the following section to ensure it is comprehensive and consistent.

Glossary and Style Guide Creation

Once translators are familiar with the content, the development team should provide the source language style guide and list of common terms, including pronunciations, for the translators to base their own target language's glossary and style guide on. If it is a sequel to another project, then the previous title's materials should be provided as reference to maintain translation continuity. If no such documents exist, then the translators should be able to create them from the source files and from the notes they took during familiarization.

A small title may require a week to name all elements, create a style guide and characterize each of the main characters or classes. Large MMORPGs could take up to 6 weeks and still not have every name finalized, but if the general policies and large chunks of each category are named and styled then the rest of translation should proceed swiftly.

The translators should brainstorm together, breaking into specialized groups on larger titles where needed, then bringing the results back to the group. That way the whole team has buy in and can see where the project as a whole is heading. Even the ideas that don't make the final cut at this phase should be kept as backups in case names need to be changed later on. You can quickly pull from the backup ideas instead of having to meet and rethink new names again. Also the reasons for naming decisions should be recorded, particularly when the team is renaming something or deliberately diverging from the original names/characterizations. It will save them having to justify their stance over and over later, and if other languages are translating via another translation (i.e. "Telephone Game" or "Chinese Whispers"-style translation) then the other languages' translators can make informed decisions on whether to stay loyal to the original source material or follow the interim language's rewrites.

Without a working glossary and style guide, it will be hard for translators to deliver consistent work. For any new names that come up after glossary creation, we suggest that the first one that comes across the name decides on a filler/working title until the group or lead can sign off on it. Then the new names should be added in an updated glossary.

Development teams must be available to answer questions on the content. The more context and answers the original creators supply, the better their work will be rendered into the target languages for a wider audience.

The style guide should list all spelling, grammar, and punctuation rules that apply to the title. If you are using a certain grammar guide or dictionary as the base for the style and spelling, then this should be listed and a copy made available to all translators to refer to during translation. For characterizations, list explanations and samples on how a character should sound, lists of word choices, and accents or speech impediments etc. (e.g. don't just say "British" or "Standard American," but list the exact time-period, region, language levels etc along with real samples and word choices. Be specific. Don't say "He sounds like Brad Pitt" but say which movie's character you are referring to.)

Updated glossary, style guide, and references (grammar guides and dictionaries) should also be provided to QA during the testing phases as the standard to test the game against.

Translation

Once the style guides and glossaries are set, then the translators can proceed to actually translate. Translators should cross-check each other before signing off on their translation. If possible, it is recommended to have specialized editors or proofreaders be the ones that do the checking to ensure consistency. Further, one linguist should always be the lead as the final word or go-to person for any decisions that need to be made.

As a general rule, from experience, a ratio of 1 editor to 3 translators for Asian languages, and 1 editor to up to 4 translators for European languages. Editors do not need to speak the source language as fluently as the translators as they should be mainly focused on the target language. However, the stronger they are at the source language, the more chance they will also find translation errors and consistency issues before QA finds them.

Each translator will have their own speed, and audio scripts will take more time than UI text as well. For planning purposes, allow for 2,000 English words (4,000 Japanese characters) per day per translator for standard text until you have your team's actual metrics in place. If there is audio involved, then expect the translators speed for the voiced sections' text may drop by as much as half, depending on how much lip-synching or time-constraint texts are involved. Further, teams will start slow before they hit their true momentum so check the speed constantly in order to see how the schedule looks.

If possible, attempt a dry run of integrating the text into the game and give the translators a chance to see their translated text on screen before the project goes to QA. Seeing the audio script's text on screen as subtitles before actual voice recording starts will allow the translators to brush up the text more before the

expense of the recording phase. This also provides early warning of text length or audio timing issues.

Voice Over Recording

If the title has audio, you will generally need to translate the recording script first, while also allowing for a pick up session at the end of the project, in order to record any new lines or address audio bugs that arise prior to localization sign off. Stage directions should also be translated (or replaced by localized instructions when needed). Be specific on what lines need to be time-synched or lip-synched to match the video.

As games become larger and more complex, the amount of audio to be recorded and tracked is becoming difficult to deal with. The development team needs to make sure they are able to manage large amounts of files in their system, and that all the final files (with unused takes removed) are provided to the translation team in an organized fashion.

For casting, do not hold back from truly localizing the characters' voices. For example, Asian languages may prefer high pitched female voices, while Western languages may prefer deeper voices, so allow each translation team make decisions on what's best for their own language rather than forcing them to all sound alike. It is rare that people compare voices across languages so it is more important that the voice works best for the target audience within itself.

A translator should always be present at the recording to help explain the material to the audio director and actors. Also on-the-spot changes will need to be made due to timing and other issues, so a translator needs to be available to make such changes. (Remember to update any as-recorded changes in the subtitles.)

To save costly mistakes, the following are recommended:

Request for Quotation

- Request quotations based on detailed scope of audio (line counts, word counts, number of actors, timing constraints per line) to enable vendors to provide accurate estimates. Quotes based on other information such as length of source language recording will add significant variance to the estimate.

Script Preparation and Translation

- Make sure your stage directions are clear and allow time/budget for them to be translated as well. (Your recording studios may not all be able to read your source language, and the translators may change the characterization of the cast, so it is wise to be aware and to allow for this.)
- Indicate which lines require Time Synching or Lip Synching before translation and deliver English audio files with the script. Time synching lines occur during scripted in game events or other areas where the dialog and game action should closely match. Lip synched lines occur during cut scenes or other areas where the speakers lips are visible. Of course it is preferred that lip movements be adjusted for each language (using such systems as FaceFX). However it is not unusual for a game to use the movie with lips animated for English lines used across all languages. Note that voice recording can take up to four times longer for lip and time synching.
- For lip synced lines deliver reference video for the translation and recording teams to match up with. If no reference video is available, the localization team can match syllables which closely resembles lip syncing.
- Indicate which lines can be recorded wildly to ensure all lines are delivered with the appropriate recording constraint (timed, lip synched or wild).
- Also, make it clear how much leeway there is between the length of the target language and the original. If all lines have to match exactly the timing or lip movements of the original, it could become expensive and take a long time to record all the audio. But if the audio's speaker doesn't appear and the timing of the lines is flexible, then that freedom will allow for faster translation and recording for

those lines.

- Make sure the file naming convention is logical so recording studios the world over aren't left searching for randomly names files when re-ordering files between characters and scenes.
- Select all takes to be used in game as soon as possible and make sure the final script and actual final takes match! (Often as-recorded changes are only implemented into the game program and not back into the original scripts, causing headaches trying to figure out which is correct.)
- Avoid re-using voice files between scenes. (Although it may work to re-use the same file in various situations in the source language, target languages may require altering translations to account for singular/plural or masculine/feminine/neutral situations.)
- Format scripts to make it obvious who the actor is speaking or interacting with. This allows studios the opportunity to re-use actors in multiple scenes or files while allowing them to avoid sequences of one actor voicing two characters that are speaking to each other.
- Do not assume source language exertions can be re-used for other languages when they do not contain distinct words. Reaction sounds vary widely between languages and cultures. Add these files to the script and ask the localization team to consider their reuse across all languages.

International Casting

- Create a casting package for the localization team comprising of character descriptions (age, sex, voice type, character background, similar voice, etc.), character image, and sample voice files.
- If any famous voice actors are used in the source language, decide which language versions require famous voice talents. This situation often arises in games affiliated with film or television shows that are localized in foreign markets. Famous voice talents are often paid a premium and will have lesser availability potentially affecting both project budget and schedule.
- Request database or live casting for key characters in the game. Database casting is much faster and cheaper and studios will typically deliver up to three voice samples for you to choose from. Have a native speaker or casting specialist review the casting sample for congruity with the character.
- Allow recording studios the latitude to cast the less important characters in the game. Due to the complexity of recording schedules and costs, this will allow them to reuse voice actors for secondary roles saving you time and money. This is one of the reasons why it is important to specify the audience in the script – to avoid a situation where one actor playing the role of two characters is speaking to himself when it is not intended.

Audio File Specifications

- Deliver volume balanced English file(s) for the international studios to normalize volume against.
- Deliver English files in the file structure you would like the international files delivered in. You may specify the studios append language codes at the end of each filename.
- Specify the format, bit rate, frequency and number of channels you wish the international recording studios to deliver their audio in.
- Specify all post-processing effects and requirements for the audio files. Some characters might need reverb or pitch-shift, others might have other special effects designed by an audio designer.
- Allow for voice files' volumes to be changed across languages. (While you can specify the general volume level, when you start dealing with large amounts of audio, it is inevitable that some files will need their volume tweaked. Better to let your audio replay system do it, than have to ask recording studios or your sound department to fix each file's volume by hand.)

Linguistic Quality Assurance

Testers should be a mix of strong linguists in the target language as well as some who are good game players. Team size will vary with the project size and schedule. While some companies allow the testers to correct bugs they find, if budget/schedule allows, then it is better if the original translators "own the translation" so that only a translator makes the changes in their section. Often there will be deliberate changes or characterizations that might be mistaken as bugs. However, the translator needs to be able to justify such deliberate changes when waiving bugs. Allow the testers to also make general suggestions (and not mix them as bugs, so more important issues can be prioritized). "Regressing" or confirming that a bug has indeed been corrected (without unexpected side-effects!) should be done more than once on separate build versions of the game. Often old text accidentally becomes implemented in a new game build, so it's always better to double check!

Also it pays to have a clear policy: for example, when QA and translators do not agree, the Lead Translator should be the one responsible for making the final call.

Master Up and Sign Off

It is recommended that a translator be present for last minute changes during the master up phase. On top of QA, one or more translators should have also played through the game to confirm that their translation has been integrated properly and all is good to go into production and release.

On top of box and docs translation, usually at this phase there will be a lot of promotional translations and guide book checking to be carried out. So keep a linguist on call even after the game has been signed off on.

Tips for MMORPGS...

Basically treat each content patch as its own localization cycle and follow the stages listed above on a smaller scale. That means the translators need to familiarize with and brainstorm the new content as they come to it.

Make sure the development team notifies the translators which sections are okay to translate, in order to avoid unnecessary rewrites. (e.g. you might use a traffic light system: Green=source language text is quite solid so it is okay to be translated; Orange=May still be minor changes, but can be translated if there is no more Green text available. Red = Translate at own risk as this source text is still not stabilized and may require rewrites.) Monitor how far the translators are behind the original source writers and make sure the original writers stop adding content in order to give both translators and QA enough time to catch up.

Translators will undoubtedly tend to specialize in certain areas of larger titles (one doing quest text or a certain regional accent, another doing item naming, another handling the interface etc). Just be aware that on long-term projects fatigue will set in, so to keep creativity and motivation up, you might allow them to rotate roles or even projects occasionally. This is where good style guides, glossaries, and transition/training periods will help you maintain consistency and quality.

PROJECT PLANNING

This section will map many of the best practices discussed previously to a typical game development process. Since processes and expectations vary widely from company to company, and methodologies employed across different platforms of distribution are ever changing, this is intended only to be a guide.

Pre-Production

- This is the stage of the project where a release plan is decided – what platforms, languages, initial project budgets, and the release date.
- Once an initial treatment document, design document, and concept art exist you can begin reviewing potential culturalization issues within the product.
- Select vendors for localization, testing, culturalization review, start getting contracts in place and putting your team together.

Production

- Kickoff the localization project – agree on a process and initial schedule, include a localization refresher for development staff if necessary. The initial schedule must include commitments to English script lockdown and English audio delivery (both to occur in this phase). The schedule should also include linguistic testing cycles and a tentative build schedule.
- Begin familiarization as well as preparing the glossary and style guide as soon as possible.
- Once character bios are available and some English audio has been recorded you can begin the casting process. Work with your localization vendor to establish the number of lines and approximate word count for each character, as well as distinguishing the key characters from the secondary ones. Allow at least 2 weeks after casting and prior to international recording for your vendor to book the talent.
- Go through script translation and international voice over recording process; allow enough time for volume balancing, post processing, and filename checks prior to implementation to the first international build.
- Begin translating the interface and in-game text. Remember most translators work at a pace of 2,000 words per day, and the later you can leave this to allow English to be edited, the fewer changes there will be in the next phase for each language.
- Work with development team to establish requirements for an international build including technology to import and export text seemlessly. Perform a dry run of a text update and new build creation.
- At the end of the production phase you should have a stable build with all or most of the localized text and audio implemented.

Alpha Phase

- Go through the internationalization testing process and ensure all localized assets are in place, all fonts are working in each language, and the user interface allows room for long translations. Some development teams create a single language build using the longest strings of the languages to create a “worst case” stress test of the user interface.
- Once you have a build in place with all internationalization bugs addressed you can begin the localization quality assurance. This will likely be the busiest phase of the project with bug fixes overlapping with new translation requests. Keeping an up to date schedule of work to be done and communicating with all of your partners will ease the friction likely to be generated during this time.

- Seek a commitment from the development team on a date to lockdown all English changes; with localization being dependant on source text, it is impossible to stop the localization process if English continues to change.
- If necessary, go through a second round of international voice over recording to address major bugs or new lines added to the script.

Beta / Localization Sign-Off / Gold Master Phase

- Once all translation requests are complete and all localization issues are addressed, grant approval to your organization to proceed with releasing the product internationally. Congratulations!
- Before too much time has elapsed, perform a brief post-mortem, and make note of ideas to be used on future projects.

APPENDIX 1 – LIST OF BEST PRACTICES

Culturalization

1. Decide on the depth of culturalization you want to achieve with your product; making it viable, legible, and/or meaningful.
2. Gain awareness of the top four cultural variables as they apply to your target markets; history, religion, ethnicity, and geopolitical imaginations.
3. Identify potential culturalization issues, assess their severity, and implement your changes to the content as early as possible in the development cycle.
4. Be precise with your content changes – make only the most minimal changes that will ensure distribution to your target markets and prevent post-release issues.

Internationalization

5. Font selection and sizing is a cooperative effort between localization and development teams.
6. Use proportional fonts with North American and European languages; Asian languages require fixed width.
7. Design the interface to allow long translations or provide a system to reposition text for each language.
8. For areas with multi-line text, enable a word wrapping system and allow text to scroll if translations are too long.
9. For areas with single line text, enable kerning to squeeze in long translations.
10. For subtitles, use a bright color san serif font with a dark outline. Position subtitles in the lower third of the screen, display for three seconds at a time, and use no more than three lines of text at a time. Indicate changes of speaker with a hyphen leading the line.
11. Store localization assets in an easy to access folder structure to enable straight forward importing and exporting.
12. User ISO 639-1 & 3166-1 language and county codes to mark assets intended for a particular locale.
13. Set aside time or budget to develop tools which will automate localization tasks.
14. For text assets, choose an encoding that supports all target languages.
15. Be prepared to swap art assets in the user interface or game world if they contain legible text.
16. Resource files should store all text used in the game; do not hardcode text strings in the source code.
17. Process resource files into separate language files so all translators can work in parallel. Indicate strings that are platform specific to ensure translations comply with console manufacturer terms.
18. For each localized build, provide a list of source language text changes.
19. XML, or one of its derivative formats, is preferred for translation work over plain text or Excel files.
20. Allow the order of variables in localized strings to be changed by language and implement a token system to ensure nouns, verbs, articles etc. can be placed in unique order by language.
21. Allow date, time, currency, and numbers to be displayed with differing figures and number separators by language.
22. Allow tables to be sorted by language as translations may change alphabetical order.
23. Implement an automatic line breaking and word wrapping feature for multiple line text boxes. Rules determining where line breaks can occur vary by language. Allow line breaking system to be overruled by use of a manual character such as a non-breaking space (0xA0).
24. Support a PAL 60Hz mode if your game is being distributed in Europe.

Localization

25. Ensure translation teams spend time familiarizing with your game prior to translation; materials to share with them include builds, story scripts, concept art, design documents, treatments or previous iterations if they exist.
26. Translators should establish a record of names and ideas as they familiarize with game content.
27. Provide the source language style guide, glossary and pronunciation of important terms to the translation team. Translation teams will develop style guides and glossaries for each target language.
28. Source language text should go through a spelling, grammar, and style check prior to sending out a translation request.
29. Plan for a translation rate of at least 2,000 words per day with an additional day for translators to request additional information when they need it.
30. Perform a dry run of integrating translations into a build prior to linguistic QA. When possible allow translators to preview subtitles prior to audio recording.
31. Try to record localized audio in one session to ensure the same audio director is available for each language and to minimize studio rental costs. Plan for an emergency pick up session towards the end of the project to fix sever bugs or cover any new source lines added to the game.
32. Provide detailed scope information to localization vendors for accurate recording quotations. At the very least they will need approximate line counts, number of actors, and type of recording constraints.
33. Audio scripts must include file names, speaker, English line, recording constraint (time synced, lip synced, or wild). Order the script chronologically so conversations can be understood in context and participants can be identified (critical for actors used to read for multiple characters).
34. Casting packages for localization vendors should include character descriptions, images, and source language voice samples.
35. Allow vendors the ability to cast unimportant characters making the recording schedule easier to manage.
36. Plan for at least 2 weeks to book actors after the international cast is chosen.
37. Specify audio file formats, file naming, folder structure, and post-processing effects you want the localization vendors to return. Also provide a volume balanced source file for the studios to normalize volume against.
38. Select a linguistic quality assurance team that possesses a mix of strong linguists in each target language as well as good game players.
39. Bugs written on localization assets should be directed to the localization vendor while bugs written on internationalization should be directed to the development team.
40. The linguistic quality assurance team should regress and confirm all bugs to be closed.
41. During the late stages of a project, keep staff on call to ensure timely resolution to any last minute bugs or content changes.
42. Once all localized content is functioning properly in-game and all localization bugs have been addressed, request the linguistic quality assurance team to sign-off on the games quality.

APPENDIX 2 - CULTURALIZATION EXAMPLES

1. History: Past and Present

1. **Age of Empires (1997)**: In *Age of Empires*, a scenario was created in which the Yamato armies of Japan invaded the Korean peninsula and effectively overwhelmed the Chosen regime of Korea. Historians tell us that this is what occurred and so the game designers diligently replicated reality. However, the government of South Korea saw history differently and disputed the scenario's accuracy and would not allow the game to be sold. A downloadable patch was developed that slightly changed the scenario so that the Yamato invasion wasn't quite as overwhelming, i.e., the Chosen armies were given a fighting chance in the game's world.
2. **Six Days in Fallujah (2009)**: This title reenacted the late 2004 events of the Second Battle of Fallujah, Iraq by the U.S. military in which U.S., British and Iraqi troops attempted to crush the insurgents using Fallujah as their base. The fight was costly for both sides and was surrounded with controversy. The coalition forces suffered over 100 deaths and over 600 wounded, while the insurgent losses were over 1,300. If several decades had passed since this event, it no doubt might make a compelling game scenario but the recent nature of the event made it so potentially inflammatory that Konami opted not to publish the title.

2. Religion and Belief Systems

1. **Kakuto Chojin (2002)**: This hand-to-hand fighting game was envisioned as a key addition to the original Xbox game library. Unfortunately, a brief audio track was incorporated into the game that included a chanted portion of the Islamic Qur'an. The error was fixed but some unfixed copies reached store shelves and the issue became widely known within a few weeks. The game was banned in Saudi Arabia and a few other Muslim countries and the backlash became so widespread that the product was globally discontinued. All of that hard work and good intentions by the development staff were essentially eliminated by a single piece of content.
2. **Resistance: Fall of Man (2006)**: When this game was released in the U.K., the Church of England was shocked to learn that their Manchester Cathedral was recreated in great detail in the game's world. But not only was it recreated but the violent action of the game played out within the church itself, something considered very disturbing by the Church of England. Sony eventually issued an apology and the Church of England subsequently issued new "Sacred Digital Guidelines" to help video game developers and others with respecting their religious structures.

3. Ethnicity and Cultural Friction

1. **Resident Evil 5 (2009)**: Even before its release, this title generated significant negative publicity due to its perceived racism. In the game, the clean cut, white Caucasian protagonist is seen roaming through a village in sub-Saharan Africa and gunning down unarmed, obviously impoverished African villagers. While the publisher Capcom was quick to show that the African villagers were infected zombies, the stark imagery of a white man killing black villagers evoked powerfully negative imagery. Notions of the "great white hunter", the "dark continent of Africa" and so forth quickly came to mind for many people. While the developers had a clear rationale for the conflict within the game's context, the backlash provided ample reason for a publisher to pause and question if mimicking that kind of negative imagery is appropriate.
2. **Pocket God (2009)**: In this iPhone game, the player is the "god" over a small fictitious island and has the ability to torment small natives though activities such as feeding them to sharks, dropping them from heights, forcing a volcano to spew hot lava on them, having killer ants devour them, and so on. The game's developer, Bolt Creative, was clear that the game is not intended to depict any specific nationality. However, with the various artifacts on the island (including a *moai* head statue specifically from Easter Island), the native outfits and their darker skin color, it was enough for Pacific

Islander advocates to complain and protest the game as a blatant use of the "primitive" ethnic stereotype.

4. Geopolitical Imaginations

1. **Hearts of Iron (2004)**: In the game *Hearts of Iron* and its sequel, the game's map was divided into somewhat arbitrary sectors similar to the classic board game *Risk*. In *Hearts of Iron*, which takes place during World War II, China was divided into several distinct pieces, including Tibet and Taiwan being portrayed as separate regions. The government of China banned the game for this very reason, even though the time context is World War II when neither Tibet nor Taiwan was part of the People's Republic of China (which didn't come into existence until 1949 and Taiwan remains apart from mainland China at present). The historical and geopolitical facts became secondary to the present government's need to reinforce their own perception of their territory in every possible context.
2. **Ghost Recon 2 (2004)**: In South Korea, the Korea Media Rating Board (KMRB; now the Game Rating Board or GRB) banned this title because the game featured a belligerent North Korean general. The South Korean government is firmly against the depiction of their northern neighbor as an aggressor because of the ongoing tensions between the two countries, as well as South Korea's long-term interest in reunification with the North. The KMRB also denied sales of the game Mercenaries in 2005 for the same reasons. In 2007, the GRB changed their minds due to local gamer pressure over freedom of speech rights and now allows titles like *Ghost Recon 2*.

APPENDIX 3 – GRAMMATICAL TOKEN EXAMPLE

Here's a more detailed explanation with actual token names (but not the source code, although it shouldn't be too hard to implement for most programmers).
Imagine we have the item table...

SING INDEF	SING DEF	SINGULAR ITEM NAME	PLR INDEF	PLR DEF	PLR ITEM NAME
a		the sword		some	the swords
an	the	axe	some	the	axes
some	the	potion	some	the	potions
the	the	Masamune	some	the	Masamune
		Excalibur		some	the Excaliburs
...	...	...	...	...	...

Then from this table, we can define the following tokens that pull from one or more columns real-time according to the context of what is happening within the game. (I am using "xxx" and "itemx" here but these could be any variable in your system.)

* <SGL_I_NAME_xxx>
  Just prints out the singular form of the item name.
  (Pretty much the default item token in all games up till now.)
  e.g. You got 1 x <SGL_I_NAME_itemx>.
  → You got 1 x sword.
* <PLR_I_NAME_xxx>
  Prints out the plural form of the item name.
  e.g. The thief stole all your <PLR_I_NAME_ itemx >!
  → The thief stole all your swords!
* <INDEF_ART_SGL_I_NAME_xxx>
  Prints out the indefinite article followed by the singular item name.
  e.g. Richard found <INDEF_ART_SGL_I_NAME_ itemx >.
  → Richard found a sword.
  → Richard found an axe.
  → Richard found the Masamune.
  → Richard found Excalibur.

* <DEF_ART_SGL_I_NAME_xxx>
  Prints out the definite article followed by the singular item name.
  e.g. Richard places <DEF_ART_SGL_I_NAME_ itemx > into the carry bag.
  → Richard places the sword into the carry bag.
  → Richard places the axe into the carry bag.
  → Richard places the Masamune into the carry bag.
  → Richard places Excalibur into the carry bag.

* <INDEF_ART_PLR_I_NAME_xxx> (...Probably not needed in E? More for FIGS.)
  Prints out the plural indefinite article followed by the plural item name.
  e.g. Richard found <INDEF_ART_PLR_I_NAME_ itemx >.
  → Richard found some swords.
  → Richard found some axes.
  (→ Richard found some Excaliburs. ...Probably never happens. Just for completeness.)

* <DEF_ART_PLR_I_NAME_xxx> (...Probably not needed in E? More for FIGS.)
  Prints out the plural definite article followed by the plural item name.
  e.g. Richard places <DEF_ART_PLR_I_NAME_ itemx > into the carry bag.
    → Richard places the swords into the carry bag.
    → Richard places the axes into the carry bag.
   (→ Richard places the Excaliburs into the carry bag. ...Probably won't occur. Just for completeness.)

* <NUM_I_NAME_xxx, val_yyy>
  Prints the number in the second parameter val_yyy, followed by
  the singular form if the number==1, otherwise the plural form.
  e.g. You buy <NUM_I_NAME_ itemx, 1>.
    → You buy 1 axe.
  e.g. You buy <NUM_I_NAME_ itemx, 2>.
    → You buy 2 axes.

* <DEF_ART_NUM_I_NAME_xxx, val_yyy>
  If the number in the parameter val_yyy == 1,
  print the singular definite article followed by the singular name form;
  otherwise print the number in val_yyy followed by the plural name form.
  e.g. Richard places <DEF_ART_NUM_I_NAME_ itemx, 1> into the carry bag.
    → Richard places the sword into the carry bag.
  e.g. Richard places <DEF_ART_NUM_I_NAME_ itemx, 5> into the carry bag.
    → Richard places 5 swords into the carry bag.
  (We could even make a similar token that instead of printing the number
  for the plural form, prints the definite plural article instead.
  e.g. "Richard places the swords into the carry bag.")

* <INDEF_ART_NUM_I_NAME xxx, val_yyy>
  If the number in the parameter val_yyy == 1,
  print the singular indefinite article followed by the singular name form;
  otherwise print the number in val_yyy followed by the plural name form.
  e.g. Richard buys <INDEF_ART_NUM_I_NAME_ itemx, 1>.
    → Richard buys a sword.
    → Richard buys an axe.
  e.g. Richard buys <INDEF_ART_NUM_I_NAME_ itemx, 5>.
    → Richard buys 5 swords.
    → Richard buys 5 axes.
  (We could even make a similar token that instead of printing the number
  for the plural form, prints the indefinite plural article instead.
  e.g. "Richard buys an axe." & "Richard buys some axes.")

* MALE/FEMALE TOKEN:
  <IF_MALE>...<ELSE_NOT_MALE>...<ENDIF_MALE>
  If the lead party member is a male,
  then it prints the former '...' part;
  otherwise it prints the latter '...' part.
  e.g.
  "Why, hello there <IF_MALE>handsome boy<ELSE>pretty girl<ENDIF>!"

...if the lead character is male then it prints...
"Why, hello there handsome boy!"
...otherwise it prints...
"Why, hello there pretty girl!"
I'm sure you'll find places where this adds character to your text, and it can be expanded to other party members if necessary, it's just that the leader of a party are usually referred to the most often in RPGs.

* SOLO/MULTI-MEMBER PARTY TOKEN:
<IF_SOLO>...<ELSE_NOT_SOLO>...<ENDIF_SOLO>
If your party only consists of one active (living) member,
then it prints the former '...' part;
otherwise it prints the latter '...' part.
In English we can probably get by with "you" as it covers both you (singular) and you (plural) situations. (Japanese, Chinese and Korean do have plural "you" though so it helps them. Some European languages change both the noun for "you" and the surrounding grammar, so this really helps them!)
Basically if your party only consists of one party member then you can do the following...
e.g. 1)
Hello <IF_SOLO>you<ELSE_NOT_SOLO>you guys<ENDIF_SOLO>!
e.g. 2)
<IF_SOLO>You should get your head fixed.<ELSE_NOT_SOLO>You people should get your heads fixed.<ENDIF_SOLO>

* SINGULAR/PLURAL TOKEN:
<IF_SING val_xxx>...<ELSE_NOT_SING>...<ENDIF_SING>
A wonderful token that can be used on anything that has a numerical value.
If val_xxx == 1, then it prints the former '...' part;
otherwise it prints the latter '...' part.
Obvious use:
"You receive <val_1> gold <IF_SING val_1>piece<ELSE_NOT_SING>pieces<ENDIF_SING>.
(Or: "You receive <val_1> gold piece<IF_SING val_1><ELSE_NOT_SING>s<ENDIF_SING>."
if you prefer brevity over clarity.)
→ "You receive 1 gold piece."
→ "You receive 2 gold pieces."
Even general statements can be improved.
e.g. if you know <val_2> contains a value representing a number of boys, say, then:
<IF_SING val_2>That boy<ELSE>Those boys<ENDIF> will never grow up.
→ That boy will never grow up. ... if <val_2> == 1;
→ Those boys will never grow up. ...otherwise.

In some games, we were even able to handle crazy sentences like...
"<Cap><ART_NUM_M_NAME_mon_no, num_of_mons> gave me <IF_SING num_of_mons>his<ELSE>their<ENDIF> <PLR_I_NAME item_no>."
...to get all manner of output from the same single line...
→ "A goblin gave me his apples."
→ "5 goblins gave me their apples."
→ "An anaconda gave me his potions."
→ "3 anacondas gave me their potions."

* DIFFERENTIATING MASCULINE/FEMININE/NEUTRAL ITEM NAMES IN GRAMMAR

<IF_NAME_xxx_M>...<IF_NAME_xxx_F>...<IF_NAME_xxx_N>...<ENDIF_NAME_xxx>

(Where xxx is the variable we are testing.)

For European languages, we will add a column next to the item names for indicating M/F/N, so we can use the token above in combination with this data to create more natural sentences.

e.g. Imagine you receive an <item_1> and want to use 'it' to refer to that item. In English that would be...

You put it in your bag.

In German we could write...

Du tust <IF_NAME_1_M>ihn<IF_NAME_1_F>sie<IF_NAME_1_N>es<ENDIF_NAME_1> in deine Tasche.

Of course, if your language only has male & female and no neutral then we would only write M or F in the column next to the name, and in the translation we'd leave that option blank in the token. e.g. in Italian...

<IF_NAME_1_M>Lo<IF_NAME_1_F>La<IF_NAME_1_N><ENDIF_NAME_1> metti nella borsa.

Similarly, we can add columns that have flags on whether these words:

- start or end with vowels or consonants
- is a proper noun or not
- are “paired items” (e.g. pairs of pants, socks or glasses)

Then we can create tokens, similar to the ones above, that branch the text real-time according to the flags pulled from the name’s row in the table, real-time. e.g.

<IF_VOWEL_x>...<ELSE_CONSONANT_x>...<END_IF_VOWEL_x>

If the first letter of the given name is a vowel,
then it prints the former '...' part;
otherwise it prints the latter '...' part.

e.g. An orb glows at <leader>'s feet.

Un orbe luit aux pieds <IF_VOWEL_HERO>d'<ELSE_CONSONANT_HERO>de <END_IF_VOWEL_HERO><HERO>.

<IF_x_PR>...<ELSE_x_NOT_PR>...<ENDIF_x_PR>

If x is a proper noun, then print former '...' part; otherwise it print the latter '...' part.

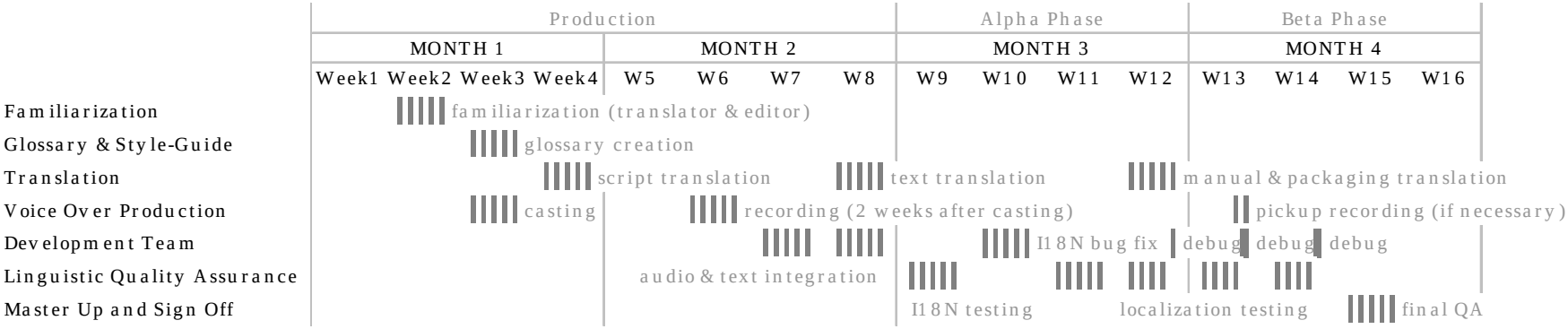
(Alternatively: If the name x has an empty entry for its corresponding definite single article (or indefinite single, for that matter) in the word table then we consider it a proper noun (i.e. an actual "character name"). So you don’t really need a flag, but it may make life easier if this rule doesn’t apply to all languages.)

We won’t go into details of German and Russian case and declension systems here, but if is basically expanding on this basic concept and adding more columns for names and articles, and then creating more tokens to access the increasing number of combinations from the table into the game text at run-time. People who are interested can enquire at the e-mail address listed at the end of this document. But be warned, these design documents can get pretty detailed so aren’t for the faint-hearted!

APPENDIX 4 – SAMPLE LOCALIZATION SCHEDULE

Schedule Parameters:

- This is a small project using one translator per language, 2 weeks worth of translation work, and 1 week worth of audio recording.
- For simplicity, regional holidays have not been accounted for.
- If translators attend recording sessions, add one week to the schedule.
- The number of translators could be increased to shorten the translation time, but it will increase familiarization and glossary creation costs; time should be balanced with costs.



CONTRIBUTORS

Stéphane Bonfils
Domhnall Campbell
Alain Dellepiane
Kate Edwards (SIG founder/chair)
Erik d'Engelbronner
Jon Fung (editor, version 2.0)
Richard Honeymoon (SIG vice-chair; author, version 1.0)
Rolf Klischewski
Víctor Alonso Lion
Eduardo Lopez
Teresa Luppino
Fabio Minazzi (SIG vice-chair)
Virginia Petrarca
Andrea Santambrogio
Miron Schmidt
Tom Slattery
Davide Solbiati
Steve Williams

Please send your additions/changes/suggestions etc. to **locsig-sc@igda.org**. Your ideas will be reviewed and implemented as soon as possible.

This is a living document that will continue to be updated as needed. While this isn't intended to be the definitive guide to game localization, it is meant to be the groundwork that will inspire people to adapt the concepts to their project's needs, and boost the overall quality of game localization worldwide. For a very in-depth look at game localization practices, we recommend *The Game Localization Handbook* (ISBN: 0763795933) by Heather Maxwell Chandler and Stephanie O'Malley-Deming (also IGDA Loc SIG members!).

If you are not a member of the IGDA Game Localization SIG, then sign up to our mailing list today via the IGDA website (igda.org) and join in the discussions!

Support your IGDA Localization SIG!