# **Localization system in Ryzom**

## Overview

There are mainly two distinct parts in localization for Ryzom. The first part (and the easiest) concerns the static localization on the client side (eg interface names, error messages). The second part is for dynamically generated text from servers.



As you can see in the diagram, there are four kind of file that makes the localization system to work. Each of this file must come in each localized language. In bold, you can see that each file contains the language code in its name.

File formats are discussed below.

## Language code

Language in Ryzom are identified by there language code as defined in [ISO 639-1](http://en.wikipedia.org/wiki/List_of_ISO_639-1_codes) plus a country code defined in [ISO 3166](http://en.wikipedia.org/wiki/ISO_3166) **if necessary**.

ISO 639-1 is a two character language code (e.g. ‘en’, ‘fr’). This is enough for most of the language we want to support.

But there is some exception, like Chinese written language.

Chinese can be written in two forms: traditional or simplified. Nonetheless, there is only one language code for Chinese: ‘hz’.

So, we must append a country code to indicate witch form of written Chinese we discuss. The language code for simplified Chinese become ‘hz-CN’ (i.e. Chinese language, Chinese country), and for traditional Chinese, it is ‘hz’ only because all other Chinese speaking country (Taiwan, Hong Kong, ? ) use the traditional Chinese.

## Identifier definition

Translated strings are associated to identifier. Identifiers are textual string that must follow the C identifier constraint with a little difference.

A C identifier must consist only of the following caracteres: ‘A-Z’, ‘a-z’, ‘0-9’, ‘@‘ and ‘\_’. Real C identifier can’t start with a number, string identifier can.

Some good identifier:

This\_is\_a\_good\_identifier

ThisIsAGoodIdentifier

\_This@is@notherGoodId

1234\_is\_a\_goodId

This\_Is\_Good\_1234

Some bad identifier:

This is a **bad** identifier

é#()|{[\_IdBAD

## File formats

There are three different translation file formats. But only two need to be learned ;-)

### Format 1

This format is used for client side static text and for server side clause text.

The file is a list of identifiant to string association (also called value string). Identifiant must conform to C identifier constraint and value string is delimited by ‘[‘ and ‘]’.

Text layout is free; you can jump line and indent as you want.

identifiant1 [textual value]

identifiant2 [other textual value]

This file can contain C style comments.

**// This is a single line comment. Continue until end of line**

identifiant1 [textual value]

**/\* This is**

**a multiline**

**comment \*/**

identifiant2 **/\* multiline comment here ! \*/** [other textual value]

Textual value can be formated for readability. New line and tab are removed in the final string value.

identifiant1 [textual

value

with

new line

and tab formating only for readability]

identifiant2 [other textual value]

If you need to specify new lines or tabulations in the value string, you must use C style escape sequence ‘\t’ for tab and ‘\n’ for new line. To write a ‘\’ in the string value, double the backslash: ‘\\’. To write a ‘]’ int the string, escape it with a backslash: ‘\]’.

identifiant1 [tabulation: **\t**This text is tabbed]

identifiant2 [New line **\n**Text on next line]

identifiant3 [Backslash: **\\**]

identifiant4 [a closing square bracket: **\]** ]

You can split the original file in multiple small file, more easy to maintain and work with.

This feature is achieved by using a C like preprocessor command “#include”.

#include "path/filename.txt"

You can have any number of include command. Included files can also contains include commands.

The path can be either an absolute path or a path relative to the location of the master file.

### Format 2

This format is used for phrases translation files.

This format is a pretty complex grammar that will be described in a near LALR syntax:

**identifiant** : [A-Za-z0-9\_@]+

**phrase** : **identifiant** ‘(‘ **parameterList** ‘)’

‘{‘

**clauseList**

‘}’

**parameterList** : **parameterList** ‘,’ **parameterDesc**

| **parameterDesc**

**parameterDesc** : **parameterType** **parameterName**

**parameterName** : **identifiant**

**parameterType** : ‘item’

| ‘place’

| ‘creature’

| ‘skill’

| ‘role’

| ‘ecosystem’

| ‘race’

| ‘brick’

| ‘tribe’

| ‘guild’

| ‘player’

| ‘int’

| ‘bot’

| ‘time’

| ‘money’

| ‘compass’

| ‘dyn\_string\_id’

| ‘string\_id’

| ‘self’

| ‘creature\_model’

| ‘entity’

| ‘bot\_name’

| ‘bodypart’

| ‘score’

| ‘sphrase’

| ‘characteristic’

| ‘damage\_type’

| ‘literal’

**clauseList** : **clauseList** **clause**

| **clause**

**clause** : **conditionList** **identifiant** **textValue**

| **identifiant** **textValue**

| **conditionList** **identifiant**

| **identifiant**

| **textValue**

**conditionList** : **conditionList** **condition**

| **condition**

**condition** : ‘(‘ **testList** ‘)’

**testList** : **testList** ‘&’ **test**

| **test**

**test** : **operand1** **operator** **reference**

**operand1** : **parameterName**

| **parameterName**’.’**propertyName**

**propertyName** : **identifiant**

**operator** : ‘=’

| ‘!=’

| ‘<’

| ‘<=’

| ‘>’

| ‘<=’

**reference** : **identifiant**

**textValue** : ‘[‘ .\* ‘]’

As in format 1, you can include C style comment in the text and indent freely and use the include command.

### Format 3: Spreadsheet unicode export

This format is the result of a Unicode text export from Spreadsheet.

Encoding should be unicode 16 bits. Columns are tab separated and rows are new line separated.

You should not write this file by hand, but only edit it with Spreadsheet.

The first row must contain the columns names.

**Info columns**

If a column name start with a ‘\*’, then all the column is ignored.

This is useful to add information column that can help translation.

**Delete character**

It is possible to insert a ‘delete’ command in the field: ‘\d’. This is useful for article translation.

Example: you have a string with the following replacement (in French):

    "Rapporte moi $item.da$ $item.name$"

And the item words file contains the following:

    item name da

    marteau marteau le

    echelle échelle l’

If the item is ‘marteau’, no problem, the replacement gives:

    "Rapporte moi **le** **marteau**"

But for the ‘echelle’, there is a supplementary space in the result:

    "Rapporte moi **l’** **échelle**"

To remove this supplementary space, you can add a ‘delete’ marker in the article definition:

    item name da

    marteau marteau le

    echelle échelle **l’\d**

This will give a correct resulting string:

    "Rapporte moi **l’échelle**"

## Working with translation files, translator point of view

###  Client side “\*.uxt” files

This file contains all static text available directly to the client. The text must conforms to format 1 described above.

There is an additional constraint: you MUST provide as a first entry the language name, as spelled in the language (eg ‘English’ for English, ‘Français’ for French).

For example, the file en.uxt must begin with:

languageName [English]

### Server side files

Server side translation is a bit more complex.

We will learn how to write server side translation in four steps (guess what: from simple to complex problem!).

**Step 1: A simple string:**

For this, you only need the phrase file.

Let’s say we want a string saying “hello world!” identified by HelloWorld.

Create a phrase entry in phrase\_en.txt:

HelloWorld ()

{

[Hello world!]

}

That’s it! No more.

Of course, you must also provide the same phrase in all the supported language, for example, in phrase\_fr.txt:

HelloWorld ()

{

[Bonjour le monde!]

}

Note that only the text value has changed. The phrase identifier MUST remain the same in all the translations files.

**Step 2: Indirection to clause\_<lang>.txt**

In step 4, we will see that the phrase file will become very complex. Thus, this file is not well fitted for giving it to a professional translator with no skill in complex grammar file. More, the complexity of the file can hide the work to do for translation.

So, you can split phrase grammar in phrase file and text value in clause file.

To do this, you must assign a unique identifier to each text value.

Let’s rebuild the previous example with indirection.

In phrase\_en.txt, create the phrase entry like this:

HelloWorld ()

{

Hello

}

We just have put an identifier in the phrase block. This means that the phrase refers to a string identified as “Hello” in the clause file.

Now, we can create the text value in clause\_en.txt:

Hello [Hello world!]

As in the first step, you must do this task for each language.

TIPS: in order to facilitate translation work, it is possible to specify the string identifier AND the string value. This can be helpful for automatic building of translation file from the original one.

Example:

HelloWorld ()

{

Hello [Bonjour le monde!]

}

In such case, the translation system always look first in the clause file and fallback to string value in the phrase file only if the string is not found in the clause file.

The other advantage is that the person who wrote the phrase file can give a simplistic version of the string that a professional translator will improve.

**Step 3: Using parameters - basics**

Here we are entering in the complex stuff!

Each phrase can receive a list of parameter.

Those parameters can be of different types:

* item,
* place,
* creature,
* skill,
* ecosystem,
* race,
* brick,
* tribe,
* guild,
* player,
* int,
* bot,
* time,
* money,
* compass,
* dyn\_string\_id,
* string\_id,
* creature\_model,
* entity,
* body\_part,
* score,
* sphrase,
* characteristic,
* damage\_type,
* bot\_name,
* literal.

Each parameter is given a name (or identifier) when declared. We will call it paramName.

Each type of parameter CAN be associated with a ‘word’ file. This file is an excel sheet (in unicode text export form) that contain translations for the parameter: its name, undefined or defined article (e.g. ‘a’, ‘the’, etc), plural name and article and any useful property or grammar element needed for translation.

The first column is very important because it associate a row of data with a particular parameter value.

Let’s begin with an example: we want to build a dynamic phrase with a variable creature race name.

First, we must build an excel sheet to define the words for creature type. This will be saved as race\_words\_<lang>.txt in unicode text export from excel. As always, you must provide a version of this file for each language.

NB: The first column MUST always be the association field and you should have a ‘name’ column as it’s the default replacement for parameter. Any other column is optional and can vary from language to language to accommodate any specific grammar constraint.

This is an example race\_words\_en.txt:

race name ia da p pia pda

kitifly Kitifly a the Kitiflys the

varynx Varynx a the Varynx the

etc…

As stated in the note above, the first column give the race identifier as defined in the game dev sheets. The second column is the ‘highly advisable’ column for the name of the race. The ‘p’ column if the plural name. ‘ia’, ‘da’ stand for indefined article and defined article.

Next, we must create a phrase with a creature parameter in phrase\_<lang>.txt:

KILL\_A\_CREATURE (**race crea**)

{}

As you can see, after the phrase identifier KILL\_THIS\_CREATURE we have the parameter list between the braket. We declare a parameter of type race named crea. Note that you choose freely your parameter name but each parameter must have a unique name (at least for one phrase).

Now, we can build the string value. To insert parameter into the string, we must specify replacement point by using the ‘$’ sign (eg $crea$) directly into the string value:

KILL\_A\_CREATURE (race crea)

{

[Would you please kill a **$crea$** for me ?]

}

As you can see, it’s not too complex. $crea$ will be replaced with the content of the field from the words file in the ‘name’ column and at the row corresponding to the race identifier.

▪ It is possible to recall any of the words file columns in the value string. We can for example dynamize the undefined article:

KILL\_A\_CREATURE (race crea)

{

[Would you please kill **$crea.ia$** **$crea$** for me ?]

}

▪ Some parameter type have special replacement rules: int are replaced with their text representation, time are converted to ryzom time readable format, as well as money.

Last but not least, the identifier and indirection rules see in step 1 and 2 are still valid.

**Step 4: Using parameters: conditional clause**

It’s time now to unveil the conditional clause system.

Let’s say that the identifier and string value we put in a phrase in the previous step is a clause. And let’s say that a phrase can contains more than one clause that can be chosen by the translation engine on the fly depending on the parameter value. This is the conditional clause system.

Let’s start a first example. As in step 3, we want to kill creature, but this time, we add a variable number of creature to kill, from 0 to n.

What we need is conditions to select between three clause: no creature to kill, one creature to kill and more than one.

First, let’s write the phrase, its parameters and the three clauses:

KILL\_A\_CREATURE (race crea**, int count**)

{

// no creature to kill

[**There is no creature to kill today.**]

// 1 creature to kill

**[Would you please kill a $crea$ for me ?]**

// more than one

[Would you please kill **$count$** $crea$ for me ?]

}

We have written down three version of the text with very different meaning and gramatical structure.

Now, add the conditions. Conditions are placed before the identifier and/or string value and are marked with bracket.

KILL\_A\_CREATURE (race crea, int count)

{

// no creature to kill

**(count = 0)** [There is no creature to kill today.]

// 1 creature to kill

**(count = 1)** [Would you please kill a $crea$ for me ?]

// more than one

**(count > 1)** [Would you please kill $count$ $crea$ for me ?]

}

Easy! no?

Now, a more complex case: we want the phrase to speak differently to male or female player. This is the occasion to introduce the self parameter. Self parameter if a ‘hidden’ parameter that is always available and that represent the addressee of the phrase.

Self parameter support name and gender property. You can provide a self\_words\_<lang>.txt file to handle special case (admin player with a translatable name for example).

Let’s rewrite the killing creature request with player gender aware style:

KILL\_A\_CREATURE (race crea, int count)

{

// -- Male player

// no creature to kill, male player

(count = 0 **& self.gender = Male**)

[Hi man, there is no creature to kill today .]

// 1 creature to kill, male player

(count = 1 **& self.gender = Male**)

[Hi man, would you please kill a $crea$ for me ?]

// more than one, male player

(count > 1 **& self.gender = Male**)

[Hi man, Would you please kill $count$ $crea$ for me ?]

// -- Female player

// no creature to kill, male player

(count = 0 **& self.gender = Female**)

[Hi girl, There is no creature to kill today.]

// 1 creature to kill, male player

(count = 1 **& self.gender = Female**)

[Hi girl, Would you please kill a $crea$ for me ?]

// more than one, male player

(count > 1 **& self.gender = Female**)

[Hi girl, Would you please kill $count$ $crea$ for me ?]

}

We have six clauses now. Three case on number of creature multiplied by two cases on the gender of the player.

As you can see, conditional test can be combined with a ‘&’ character. This means that all the tests must be valid to select the clause.

You can use any parameter as left operand for the test. You can also specify a parameter property (coming from words file) as operand.

On the other hand, right test operand must be constant value (either textual or numerical value).

Available operators are =, !=, <, <=, >, >=.

In some case, you could need to make OR’ed test combination. This is possible by simply specifiying multiple condition list before a clause:

FOO\_PHRASE (int c1, int c2)

{

**(c1 = 0 & c2 = 10)**

**(c1 = 10 & c2 = 0)**

[This clause is selected if :

c1 equal zero **and** c2 equal ten

**or**

c1 equal ten **and** c2 equal zero]

}

▪ Detailed clause selection rules:

* A valid clause is a clause where the conditional combination are true with a given set of parameter value.
* Clauses are evaluated in the same order as they are written in the phrase file. The first valid clause will be selected.
* If the first clause doesn’t have condition, it is used as a fallback clause when none of the conditional clauses are selected.
* If there is no fallback clause and none of the clauses are selected, then the first clause is selected.
* For the same phrase, each language can provide its own set of condition and clause to accommodate its gammatical needs.

###  Harcoded parameters properties reference

Here you file find an exhaustive list of harcoded parameters properties.

These properties are always available even if no words files are provided.

Futhermore, the harcoded property can’t be substituted by a word file specifying a column of the same name.

|  |  |
| --- | --- |
| **Parameter** | **Property** |
| Item |  |
| Place |  |
| Creature | name : model name of the creaturegender : gender of the creature from the model |
| Skill |  |
| Ecosystem |  |
| Race |  |
| Brick |  |
| Tribe |  |
| Guild |  |
| Player | name : name of the playergender : gender of the player in the mirror |
| Bot | career : the career of the botrole : the role of the bot as defined in creature.basics.chatProfilename : name of the creaturegender : gender of the creature from the model |
| Integer |  |
| Time |  |
| Money |  |
| Compass |  |
| dyn\_string\_id | Only != and == test available. Mainly to compare parameter with 0. |
| string\_id | Only != and == test available. Mainly to compare parameter with 0 |
| self | name : name of the playergender : gender of the player in the mirror |
| creature\_model | NB : use the creature\_words translation file ! |
| entity | == 0, != 0: test if the entity is Unknown or not.name : name of the creature or name of the player.gender : gender of the creature from the model or gender of the player (from the player info). |
| bodypart |  |
| score |  |
| sphrase |  |
| characteristic |  |
| damage\_type |  |
| bot\_name |  |

## Translation workflow

In the following translation workflow, we consider that the reference language is English.

There is a series of tools and bat file to help in getting translation in sync. Here is a step by step description of how to work on translation file, what tool to use and when.

Only addition to existing translation file is supported by the translation tools. If you need to modify or remove existing translation string or phrase structure, this must be done ‘by hand’ with a maximum of attention to all the languages versions.

In most case, it is better to create a new translation entry instead of dealing with a modification in all the translations files and it’s almost safe to leave old unused string in the files.

At least, you should NEVER make modification when there are pendings diff file.

**It is highly advisable to strictly respect the described workflow** in order to avoid translation problems, missing strings and other weird problems that can arise from working with many language version of a set of file.

Translation work is done in cooperation between the publisher who is doing the ‘technical’ part of the task and a profesionnal translator contractor who only translate simple string into a very high quality and rich string with respect to the context.

The tools that generate diff file for translation keep the comments from the reference version. This can be helpful to give additionnal information to the translator about the context of the text.

Moreover, for phrase files, the diff file automaticaly include comments that describe the parameter list.

### Translation repository structure

All files to translate are stored in a well defined directory structure called ‘the translation repository’. All translation work is done in this repository.

Tools are provided to install the translated file in the client and server repository after a translation cycle is done.

translation/ Root directory for translation repository

languages.txt A simple text file that contains all the language code (ISO 639-2) that we are working on (eg : en, fr, etc…).

work/ This is the starting point for addition of new content. This directory contains any files that can be edited for addition.

diff/ Contains all diff files generated by tools. This is where most of the work is done. After the merge operation is applied to integrate translated diff into translated file, the diff file are moved to the history directory.

When diff files are generated, they are prefixed with a version number automaticaly computed from the current Unix time. This permits to generate new diff file without waiting for diff translation to be done. More, when you generate new diff file with existing none merged diff, new diff only contains the new difference (is it clear or what? ☺ ).

history/ Contains all the diff files that have been translated and merged. This is for backup and security reason.

translated/ Contains all translated file. The content of this directory is installed into client and server repository when translation cycle is done. You should never change anything by hand in this directory unless you now exactly what you are doing.

### Client side static string

**Initial task:**

Publisher

* Creates the reference file en.uxt in the work directory, fill in the required first string ‘LanguageName’ then add any number of string for the English version.
* Generates static string diff files with the command make\_string\_diff.bat. This will create diff files for each language in the diff directory.
* Send the diff files to the Translator.

Translator

* Translate the diff files.
* Send them back to the Publisher.

Publisher

* Put the translated diff files in the diff directory (this will overwrite non translated diff files with translated one).
* Merge the translated diff files with the command merge\_string\_diff.bat. This will create the <lang>.uxt files for each language in the translated directory and move the diff in the history directory.

After the initial task is done, the workflow enters the in incremental mode.

**Incremental task:**

Publisher

* Add string in the reference file en.uxt in the addtion directory.
* Generates static string diff files with the command make\_string\_diff.bat. This will create diff files for each language in the diff directory.
* Send the diff files to the Translator.

Translator

* Translate the diff files.
* Send them back to the Publisher.

Publisher

* Put the translated diff files in the diff directory (this will overwrite non translated diff files with translated one).
* Merge the translated diff files with the command merge\_string\_diff.bat. This will apply the diff files to the <lang>.uxt files for each language in the translated directory and move the diff in the history directory.

### Server side dynamic string

**Initial task:**

Publisher

* Creates the reference file phrase\_en.uxt in the addition directory and add any number of phrase entries for the English version.
* Generates phrase diff files with the command make\_phrase\_diff.bat. This will create diff files for each language in the diff directory.
* Translate the phrase diff file. This implie good knowledge of gramatical structure of each language to accommodate clause selection rules.
* Merge the translated diff with the command merge\_phrase\_diff.bat. This will create all the phrase\_<lang>.txt in the translated directory from the translated diff then move diff files to history directory.
* Generates clauses diff file with the command make\_clause\_diff.bat. This create clause diff file in the diff directory for all language.
* Send the clause diff files to the Translator

Translator

* Translate the clause diff files.
* Send them back to the Publisher.

Publisher

* Put the translated clause diff files in the diff directory (this will overwrite non translated clause diff files with translated one).
* Merge the translated clause diff files with the command merge\_clause\_diff.bat. This will create the clause\_<lang>.txt files for each language in the translated directory and move the diff in the history directory.

After the initial task is done, the workflow enters the in incremental mode.

**Incremental task:**

Publisher

* Add new phrase in phrase\_en.uxt in the addition directory.
* Generates phrase diff files with the command make\_phrase\_diff.bat. This will create diff files for each language in the diff directory.
* Translate the phrase diff file. This implie good knowledge of gramatical structure of each language to accommodate clause selection rules.
* Merge the translated diff with the command merge\_phrase\_diff.bat. This will append the diff to there repective phrase\_<lang>.txt file in the translated directory then move diff files to history directory.
* Generates clauses diff file with the command make\_clause\_diff.bat. This create clause diff file in the diff directory.
* Send the clause diff files to the Translator

Translator

* Translate the clause diff files.
* Send them back to the Publisher.

Publisher

* Put the translated clause diff files in the diff directory (this will overwrite non translated clause diff files with translated one).
* Merge the translated clause diff files with the comment merge\_clause\_diff.bat. This will append the diff file to there repective clause\_<lang>.txt files for each language in the translated directory and move the diff in the history directory.

### Server side words files

NB: ‘words’ has nothing to do with the microsoft program Word!

Words files are always updated by hands because they are rarely updated by the publisher (and generaly, this will be for big update). Moreover, when new phrase are translated, it can be necessary to create and fill new column in one of the words file to accommodate the translation.

So, there is only a workflow but no tools.

**Initial task:**

Publisher

* Build the initial sheet for each type of parameter with all the possible identifier in the reference language in the translated directory.
* Create and fill the default ‘name’ and ‘p’ columns.
* Create the sheets for all languages by copying the reference language sheets.
* Create and fill any basic column depending on languages.
* Sends all the sheets to the Translator.

Translator

* Translate all the sheets in all language, eventualy, create any needed columns.
* Send back the translated sheets to the Publisher.
* Keep a copy of the translated sheets as reference for phrase and clause translation.

Publisher

* Put the translated sheets in the translated directory, overwriting old ones.

After this initial task, there is two possible events:

**New column(s) needed:**

Translator

* While translating phrase or clause diff, it comes that one or more new columns are needed for some language and parameter type.
* Define the needed columns
* Contact Publisher to check that no sheets updates are pending. If yes, first apply the ‘new sheets content’ workflow.
* Add and fill the new columns into the concerned sheets/langs.
* Send the sheets to the Publisher.

Publisher

* Put the new sheets in the translated directory, overwriting old ones.

**New sheets content:**

Publisher:

* New game content is to be integrated in the game.
* Contact Translator to check that no sheets updates are pending. If yes, first apply the ‘new column(s) needed’ workflow.
* Create new sheets for the reference language and containing only the new content.
* Add and fill the default columns in the new sheets (see ‘Initial task’).
* Create new sheets for all the languages by copying the reference languages sheets.
* Add but DON’T FILL all the columns to repect the current sheet format for each type, each language.
* Send the new sheets to the Translator.

Translator:

* Translate the new sheets.
* Append the new sheets at the end of the existing sheets.
* Send the merge result to the Pushisher.
* Keep the merge result as reference for phrase and clause translation and futur content addition.

Publisher:

* Put the new sheets in the translated directory, overwriting old ones.

**Installing the translated files**

After a cycle of translation is terminated, you must install the translated files into the client and servers directory strucure.

This is done via the command intall\_translation.bat.

The <lang>.uxt file are copied into the client strucure in Ryzom/gamedev/language.

All the other files are copied in Ryzom/data\_shard.

To apply client side translation, Publisher needs to make a patch.

To apply server side translation, just enter the command ‘reload’ on the InputOutputService.

## Working with translation files: programmer view

As a NeL/Ryzom programmer, you can use the translation system with a very few calls.

###  Accessing client static strings

To obtain a unicode string from a string identifier, you use the NLMISC::CI18N class.

First of all, you must ensure that the translation file \*.uxt are available in the search path.

Then, you can load a language string set by calling NLMISC::CI18N::load(languageCode).

The parameter languageCode is the language code as defined in chapter 2 “Language code”.

After that, call the method NLMISC::CI18N::get(identifier) to obtain the unicode string associated to identifier.

### Dynamic strings

Dynamic string requires a bit more work and a complete shard infrastructure.

Dynamic string management involves a requesting service (RQS), the InputOutputService (IOS), the FrontEnd (FE), the ryzom client plus the basic services to run the other (naming, tick, mirror).



RQS is a service that wants to send a dynamic string to the client.

RQS also send the dynamic string identifier to the client by using the database or any other way.

The proxy is a small piece of code that builds and sends the PHRASE message to IOS service.

IOS make the big task of parsing parameter, selecting the good clause and building the resulting string then sending a minimum amount of data to the client.

The client receives the phrase and requests any missing string element to the IOS.

**Building parameter list and sending the string**

To access the proxy function, you need to include game\_share/string\_manager\_sender.h and link with game\_share.lib.

You first need to build the parameters list. This is done by filling a vector of STRING\_MANAGER::TParam struture. For each parameter, you must set the Type field then write the appropriate member data.

You MUST exactly respect the phrase parameter definition in the phrase file.

Then, you can call

uint32 STRING\_MANAGER::**sendStringToClient**(

NLMISC::CEntityId destClientId,

const std::string &phraseIdentifier,

const std::vector<STRING\_MANAGER::TParam> parameters)

destClientId is the entity id of the destination client, phraseIdentifier is the indentifier like writen in the phrase file, parameters is the vector of parameter you have build before.

This function returns the dynamic ID that is assigned to this phrase.

Example : sending the ‘kill a creature’ phrase (see § 5.2, step 4) :

// include the string manager proxy definition

#include “game\_share/string\_manager\_sender.h”

uint32 killCreatureMessage(

EntityId destClient,

GSPEOPLE::EPeople race,

uint32 nbToKill)

{

std::vector<STRING\_MANAGER::TParam> params;

STRING\_MANAGER::TParam param;

// first, we need the creature race

param.Type = STRING\_MANAGER::creature;

param.Enum = race;

params.push\_back(param);

// second, the number of creature to kill

param.Type = STRING\_MANAGER::integer;

param.Int = nbToKill;

params.push\_back(param);

// and now, send the message

uint32 dynId = STRING\_MANAGER::sendStringToClient(

destClient,

“KILL\_A\_CREATURE”,

params);

return dynId;

}

**Member to fill in TParam depending on the type of parameter:**

item: Fill SheetId with the sheet id of the item

place: Fill Identifier string with the place identifier

creature: Fill EId with the creature entity id

skill: Fill Enum with the enum value from SKILLS::ESkills

ecosystem: Fill Enum with the enum value from ECOSYSTEM::EEcosystem

race: Fill Enum with the enum value from GSPEOPLE::EPeople

brick: Fill SheetId with the sheet id of the brick

tribe: not defined yet

guild: not defined yet

player: Fill EId with the player entity id

bot: Fill EId with the bot entity id

integer: Fill Int with the integer value (sint32)

time: Fill Time with the time value (uint32)

money: Fill Money with the money value (uint64)

compass: not defined yet

dyn\_string\_id: Fill StringId with a dynamic string id

string\_id: Fill StringId with a string id

creature\_model: Fill SheetId with the sheet id of the creature

entity: Fill EId with the creature,npc or player entity

body\_part: Fill Enum with the enum value from BODY::TBodyPart

score: Fill Enum with the enum value from SCORES::TScores

sphrase: Fill SheetId with the sheet id of the phrase

characteristic: Fill Enum with the enum value from CHARACTERISTICS::TCharacteristic

damage\_type: Fill Enum with the enum value from DMGTYPE::EDamageType

bot\_name: Fill Identifier with the bot name without function.

literal: Fill Literal with the Unicode literal string.

**Accessing the dynamic string from the client**

On the client side, accessing the dynamic string is pretty easy. You only need to take care of transmission delay in certain case.

Once you get the dynamic string id from anyway (eg database), you just need to pol the getDynString method of STRING\_MANAGER::CStringManagerClient.

The method return false until the requested string is incomplete or unknown.

Even when the method return false, it could return a partial text with missing replacement text.

Dynamic strings are dynamicaly stored, and if your code is smart enough, it could release the dynamic string memory when they are no more needed by calling releaseDynString.

In order to be efficient, one can call each frame the getDynString method until it return true, then store away the string and call the releaseDynString.

STRING\_MANAGER::CStringManagerClient is based on the singleton pattern so you must call STRING\_MANAGER::CStringManagerClient::instance() to get a pointer to the singleton instance.

Here is a simple code sample.

// include the string manager client definition

#include “string\_manager\_client.h”

using namespace STRING\_MANAGER;

/\*\* A method that receive the dynamic string id

\* and print the dynamic string in the log.

\* Call it each frame until it return true.

\*/

bool foo(uint32 dynStringId)

{

ucstring result;

bool ret;

CStringManagerClient \*smc = CStringManagerClient::instance();

ret = smc->getDynamicString(dynStringId, result)

if (!ret)

nlinfo(“Incomplete string : %s”, result.toString().c\_str());

else

{

nlinfo(“Complete string : %s”, result.toString().c\_str());

// release the dynamic string

smc->releaseDynString(dynStringId);

}

return ret;

}

# Ryzom text creation guide

There are many place for text in Ryzom, this page will clarify the text identification conventions, the available context for text insertion and contextual text parameters.

## 1.     Identifier conventions

### 1.1. Strings identifiers in **en.uxt**

These identifiers are written lower case with capital at start of each new word.

Example:

     aSimpleIdentifier

     anotherIdentifier

### 1.2. Phrases identifiers in phrase\_en.txt

These identifiers are written in capitals, words are separated with an underscore.

Example:

     A\_SIMPLE\_IDENTIFIER

     ANOTHER\_IDENTIFIER

### 1.3. Strings (or clauses) identifiers in clause\_en.txt

These identifiers are written like strings identifiers in en.uxt.

But, as they are inside a phrase definition, they must contains the name of the phrases as base name. The phrases name is lowered to respect the string identifiers convention.

Example:

     In a phrase named A\_SIMPLE\_LABEL, the identifier should be

     aSimpleLabel

Furthermore, when there is more than one clause for a given phrase, the clause identifier must be followed by some tags that give clues to the translator about the meaning of each clause.

Example:

            In a phrases named A\_SIMPLE\_LABEL and that contains two clauses, one for singular, the other for plural, we could have the following two identifiers:

     aSimpleLabelS

     aSimpleLabelP

## 2.     Text contexts

### 2.1. Chat contexts

Chat context cover any texts that come from an NPC through the chat windows and text bubbles.

#### 2.1.1- Bot says/shout around

There is only one parameter available: the npc entity that say/shout.

Phrase name start with SAY\_

Phrase sample:

     SAY\_XXX (bot b)

{

     sayXxx    [Hello there, my name’s $b$, anybody hear me?]

}

#### 2.1.2- Bot talk to escort leader (a player)

Two parameters: the bot that talk and the player.

Phrase name start with TALK\_

Phrase sample:

     TALK\_XXX (bot b, player p)

{

     talkXxx   [Hello $p$, my name’s $b$, I need help !]

}

#### 2.1.3- Bot says/shout in response to target click

Two parameters: the bot clicked and the player.

Phrase name start with CLICK\_

Phrase sample:

     CLICK\_XXX (bot b, player p)

{

     clickXxx  [Hello $p$, my name’s $b$, did you click me ?]

}

### 2.2. Interactive context (**aka** **botchat**)

Botchat cover any texts that come in the interactive NPC dialog.

#### 2.2.1- Mission related phrases

Static missions

All phrase name related to mission have of root defined by the name of the mission as placed in the world editor mission node.

From this root, there are several extension that must be appended to form the phrase names:

\_TITLE                        for mission title

\_STEP\_X        for step X text (mission step start from 1)

\_END              for the mission end text.

Example:

Given a mission named                          INSTRUCTOR\_MIS\_1,

The mission title will be             INSTRUCTOR\_MIS\_1\_TITLE,

Step 1 mission text will be         INSTRUCTOR\_MIS\_1\_STEP\_1,

Step 2 mission text will be         INSTRUCTOR\_MIS\_1\_STEP\_2,

Mission end text will be             INSTRUCTOR\_MIS\_1\_END

Parameters:

XXXXXXX\_TITLE (bot b, player p)

    -b est le bot à qui le joueur parle

    -p est le joueur

XXXXXXX\_STEP\_X (bot giver, bot current, bot target, bot previous, player p)

    - giver est le donneur de la mission

    - current est le bot à qui le joueur parle

    - target est le bot à aller voir pour la prochaine etape

    - previous est le bot vu à l'étape précédente

    - p est le joueur

XXXXX\_END (bot current, bot giver, bot previous, player p)

    - giver est le donneur de la mission

    - current est le bot à qui le joueur parle

    - previous est le bot vu à l'étape précédente

    - p est le joueur

Les paramètres des textes d'étapes du journal dépendent de la nature de l'étape (voir avec Nicolas Brigand).

Pour les textes de progression de mission dans le menu contextuel, il en existe deux :

MISSION\_STEP\_GIVE\_ITEM\_CONTEXT (bot giver, bot previous, item i)

MISSION\_STEP\_TALK\_CONTEXT (bot giver, bot previous)

Le premier est le texte standard, le second est affiché quand on doit donner quelque chose au bot.

#### 2.2.2- Additional context menu entry

It is possible to add simple ‘informational’ entry in the bot contextual menu. This item is composed of two phrases: the menu entry display name and the text content displayed after clicking the menu entry.

Two parameters: the bot supporting the context menu and the player.

Phrase name start with BC\_MENU\_ for menu entry and BC\_MENUTEXT\_

### 2.3.   System message (e.g. combat info)

Parameters are phrases dependent, but there are some well defined phrases types:

COMBAT\_

MAGIC\_

HARVEST\_

CRAFT\_

DEATH\_

PROGRESS\_

### 3.4. None of the above…