Localization tutorial

Summary

This is our first basic C++ tutorial. It also shows how the localization module (`orxLOCALE`) works.

See previous basic tutorials for more info about basic object creation, clock handling, frames hierarchy, animations, cameras & viewports, sounds & musics, FXs, physics and scrolling.

This code is a basic C++ example to show how to use orx without writing C code.
This tutorial could have been architectured in a better way (cutting it into pieces with headers files, for example) but we wanted to keep a single file per *basic* tutorial.

This stand alone executable also creates a terminal console [1], but you can have you own console-less program if you wish.
For visual studio users (windows), it can easily be achieved by writing a `WinMain()` function instead of `main()`, and by calling `orx_WinExecute()` instead of `orx_Execute()`.

This tutorial simply display orx's logo and a localized legend. Press space or click left mouse button to cycle through all the available languages for the legend's text.

Some explanations about core elements that you can find in this tutorial:

- `Run function`: Don't put *ANY* logic code here, it's only a backbone where you can handle default core behaviors (tracking exit or changing locale, for example) or profile some stuff. As it's directly called from the main loop and not part of the clock system, time consistency can't be enforced. For all your main game execution, please create (or use an existing) clock and register your callback to it.
- `Event handlers`: When an event handler returns orxSTATUS_SUCCESS, no other handler will be called after it for the same event. On the other hand, if orxSTATUS_FAILURE is returned, event processing will continue for this event if other handlers are listening this event type. We'll monitor locale events to update our legend's text when the selected language is changed.
- `orx_Execute()/orxWinExecute()`: Inits and executes orx using our self-defined functions (Init, Run and Exit). We can of course not use this helper and handles everything manually if its behavior doesn't suit our needs. You can have a look at the content of `orx_Execute()/orx_WinExecute()` [2] to have a better idea on how to do this.

Details

Let's start with the includes.

```
#include "orx.h"
```

That's all one need to include so as to use orx. This include works equally with a C or a C++ compiler [3].

Let's now have a look at our Game class that contains orx's `Init()`, `Run()` and `Exit()` callbacks.

```
class Game
{
public:
  static orxSTATUS orxFASTCALL  EventHandler(const orxEVENT *_pstEvent);
  static orxSTATUS orxFASTCALL  Init();
  static void orxFASTCALL       Exit();
  static orxSTATUS orxFASTCALL  Run();

  void SelectNextLanguage();

  Game() : m_poLogo(NULL), s32LanguageIndex(0) {};
  ~Game() {};

private:
  orxSTATUS                     InitGame();

  Logo  *m_poLogo;
  orxS32 s32LanguageIndex;
};
```

All the callbacks could actually have been defined out of any class. This is done here just to show how to do it if you need it.
We see that our Game class also contains our Logo object and an index to the current selected language.

Let's now have a look to our Logo class definition.

```
class Logo
{
private:
  orxOBJECT *m_pstObject;
  orxOBJECT *m_pstLegend;

public:
  Logo();
  ~Logo();
};
```

Nothing fancy here, we have a reference to an `orxOBJECT` that will be our logo and another one that will be the displayed localized legend.
As you'll see we won't use the reference at all in this executable, we just keep them so as to show a proper cleaning when our Logo object is destroyed. If we don't do it manually, orx will take care of it when quitting anyway.

Let's now see its constructor.

```
Logo::Logo()
{
  m_pstObject = orxObject_CreateFromConfig("Logo");
  orxObject_SetUserData(m_pstObject, this);

  m_pstLegend = orxObject_CreateFromConfig("Legend");
}
```

As seen in the previous tutorials we create our two objects (Logo and Legend) and we link our Logo C++ object to its orx equivalent using `orxObject_SetUserData()`.

```
Logo::~Logo()
{
  orxObject_Delete(m_pstObject);
  orxObject_Delete(m_pstLegend);
}
```

Simple cleaning here as we only delete our both objects.

Let's now see our main function.

```
int main(int argc, char **argv)
{
  orx_Execute(argc, argv, Game::Init, Game::Run, Game::Exit);

  return EXIT_SUCCESS;
}
```

As we can see, we're using the `orx_Execute()` helper that will initialize and execute orx for us.
In order to do so, we need to provide it our executable name and the command line parameters along with three callbacks: `Init()`, `Run()` and `Exit()`.
We will only exit from this helper function when orx quits.

Let's have a quick glance at the console-less version for windows.

```
#ifdef __orxMSVC__

int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance, LPSTR
lpCmdLine, int nCmdShow)
{
  // Inits and executes orx
  orx_WinExecute(Game::Init, Game::Run, Game::Exit);

  // Done!
  return EXIT_SUCCESS;
}

#endif
```

Same as for the traditional `main()` version except that we use the `orx_WinExecute()` helper that will compute the correct command line parameters and use it. [4]
This only works for a console-less windows game [5].

Let's now see how our `Init()` code looks like.

```
orxSTATUS Game::Init()
{
  orxLOG("10_Locale Init() called!");

  return soMyGame.InitGame();
}
```

We simply initialize our Game instance by calling its `InitGame()` method.
Let's see its content.

```
orxEvent_AddHandler(orxEVENT_TYPE_LOCALE, EventHandler);

m_poLogo = new Logo();

std::cout << "The available languages are:" << std::endl;
for(orxS32 i = 0; i < orxLocale_GetLanguageCounter(); i++)
{
  std::cout << " - " << orxLocale_GetLanguage(i) << std::endl;
}

orxViewport_CreateFromConfig("Viewport");
```

We simply register a callback to catch all the `orxEVENT_TYPE_LOCALE` events.
We then instanciate our Logo object that contains both logo and legend.
We also outputs all the available languages that have been defined in config files. We could have used the `orxLOG()` macro to log as usual (on screen and in file), but we did it the C++ way here to show some diversity.
We finish by creating our viewport, as seen in all the previous tutorials.

Let's now see our `Exit()` callback.

```
void Game::Exit()
{
  delete soMyGame.m_poLogo;
  soMyGame.m_poLogo = NULL;

  orxLOG("10_Locale Exit() called!");
}
```

Simple Logo object deletion here, nothing surprising.

Now let's have a look to our `Run()` callback.

```
orxSTATUS Game::Run()
{
  orxSTATUS eResult = orxSTATUS_SUCCESS;

  if(orxInput_IsActive("CycleLanguage") &&
orxInput_HasNewStatus("CycleLanguage"))
  {
    soMyGame.SelectNextLanguage();
  }

  if(orxInput_IsActive("Quit"))
  {
    orxLOG("Quit action triggered, exiting!");
    eResult = orxSTATUS_FAILURE;
  }

  return eResult;
}
```

Two things are done here.
First when the input `CycleLanguage` is activated we switch to the next available language, then when the `Quit` one is activated, we simply return `orxSTATUS_FAILURE`.
When the `Run()` callback returns `orxSTATUS_FAILURE` orx (when used with the helper `orx_Execute()`) will quit.

Let's have a quick look to the `SelectNextLanguage()` method.

```
void Game::SelectNextLanguage()
{
  s32LanguageIndex = (s32LanguageIndex == orxLocale_GetLanguageCounter() -
1) ? 0 : s32LanguageIndex + 1;

  orxLocale_SelectLanguage(orxLocale_GetLanguage(s32LanguageIndex));
}
```

We basically go to the next available language (cycling back to the beginning of the list when we reached the last one) and selects it with the `orxLocale_SelectLanguage()` function.
When doing so, all created `orxTEXT` objects will be automatically updated if they use a localized string. We'll see how to do that below in the config description.
We can also catch any language selection as done in our `EventHandler` callback.

```
orxSTATUS orxFASTCALL Game::EventHandler(const orxEVENT *_pstEvent)
{
  switch(_pstEvent->eID)
  {
    case orxLOCALE_EVENT_SELECT_LANGUAGE:

      orxLOCALE_EVENT_PAYLOAD *pstPayload;
      pstPayload = (orxLOCALE_EVENT_PAYLOAD *)_pstEvent->pstPayload;
      orxLOG("Switching to '%s'.", pstPayload->zLanguage);
      break;

    default:

      break;
  }
  return orxSTATUS_FAILURE;
}
```

As you can see, we only track the `orxLOCALE_EVENT_SELECT_LANGUAGE` event here so as to display which is the new selected language.

We're now done with the code part of this tutorial. Let's now have a look at the config.

First let's define our display.

```
[Display]
ScreenWidth    = 800
ScreenHeight   = 600
Title          = Stand Alone/Locale Tutorial
```

As you can see, we're creating a window of resolution 800×600 and define its title.

Let's now define our resource paths.

```
[Resource]
Texture = ../data/object
```

We're only using textures and they're all in a single folder (../data/object).

We now need to provide info for our viewport and camera.

```
[Viewport]
Camera          = Camera
BackgroundColor = (20, 10, 10)

[Camera]
FrustumWidth  = @Display.ScreenWidth
FrustumHeight = @Display.ScreenHeight
FrustumFar    = 2.0
Position      = (0.0, 0.0, -1.0)
```

Nothing new here as everything was already covered in the viewport tutorial.

Let's now see which inputs are defined.

```
[Input]
SetList = MainInput

[MainInput]
KEY_ESCAPE  = Quit
KEY_SPACE   = CycleLanguage
MOUSE_LEFT  = CycleLanguage
```

In the `Input` section, we define all our input sets. In this tutorial we'll only use one called `MainInput` but we can define as many sets as we want (for example, one for the main menu, one for in-game, etc…).

The `MainInput` sets contain 3 mapping:

- KEY_ESCAPE will trigger the input named `Quit`
- KEY_SPACE and MOUSE_LEFT will both trigger the input named `CycleLanguage`

We can add as many inputs we want in this section and bind them to keys, mouse buttons (including wheel up/down), joystick buttons or even joystick axes.

Let's now see how we define languages that will be used by the `orxLOCALE` module.

```
[Locale]
LanguageList = English # French # Spanish # German # Finnish # Swedish #
Norwegian # Chinese

[English]
Content       = This is orx's logo.
Lang          = (English)

[French]
Content       = Ceci est le logo d'orx.
Lang          = (Français)
LocalizedFont = CustomFont

[Spanish]
Content       = Este es el logotipo de orx.
Lang          = (Español)

[German]
Content       = Das ist orx Logo.
Lang          = (Deutsch)
LocalizedFont = CustomFont

[Finnish]
Content       = Tämä on orx logo.
Lang          = (Suomi)

[Swedish]
Content       = Detta är orx logotyp.
Lang          = (Svenska)
LocalizedFont = CustomFont

[Norwegian]
Content       = Dette er orx logo.
Lang          = (Norsk)

[Chinese]
Content       = 这是Orx的标志
Lang          = (Chinese)
LocalizedFont = CustomChineseFont
```

To define languages for localization we only need to define a `Locale` section and define a `LanguageList` that will contain all the languages we need.
After that we need to define one section per language and for every needed keys (here `Content` and `Lang`) we set their localized text.
In the same way, we defined `LocalizedFont` for one language out of two, and we will use it for specifying a specific font based on the text/language combination.

As the localization system in based on orx's config one, we can use its inheritance capacity for easily adding new languages to the list (in another extern file, for example), or even for completing languages that have been partially defined.

Let's now see how we defined our Logo object.

```
[LogoGraphic]
Texture = orx.png
Pivot   = center

[Logo]
Graphic   = LogoGraphic
FXList    = FadeIn # LoopFX # ColorCycle1
Smoothing = true
```

Again, everything we can see here is already covered in the object tutorial.
If you're curious you can look directly at 10_Locale.ini to see which kind of FXs we defined, but we won't cover them in detail here.

Next thing to check: our Legend object.

```
[Legend]
ChildList = Legend1 # Legend2
```

Surprise! Actually it's an empty object that will spawn two child objects: `Legend1` and `Legend2`. 😃

Code-wise we were creating a single object called `Legend` but apparently we'll end up with more than one object.
The same kind of technique can be used to generated a whole group of objects, or a complete scenery for example, without having to create them one by one code-wise.
It's even possible to chain objects with `ChildList` and only create a single object in our code and having hundreds of actual objects created.
However, we won't have direct pointers on them, which means we won't be able to manipulate them directly.
That being said, for all non-interactive/background object it's usually not a problem.
Be also aware that their frames (cf. frame tutorial) will reflect the hierarchy of the `ChildList` 'chaining'.

Ok, now let's get back to our two object, `Legend1` and `Legend2`.

```
[Legend1]
Graphic       = Legend1Graphic
```

```
Position      = (0, 0.25, 0.0)
FXList        = ColorCycle2
ParentCamera  = Camera

[Legend2]
Graphic       = Legend2Graphic
Position      = (0, 0.3, 0.0)
FXList        = @Legend1
ParentCamera  = @Legend1
```

They look very basic, they're both using the same FX (`ColorCyle2`), they both have a `Position` and each of them has its own `Graphic`.

NB: We can also see that we defined the `ParentCamera` attribute for both of them. This means that their actual parent will become the camera and not the `Legend` object in the end.
However `Legend` will still remain their owner, which means that they'll automatically be erased when `Legend` will be deleted.

Let's now finish by having a look at their `Graphic` objects.

```
[Legend1Text]
String = $Content
Font   = $LocalizedFont

[Legend2Text]
String = $Lang

[Legend1Graphic]
Pivot = center
Text  = Legend1Text

[Legend2Graphic]
Pivot = center
Text  = Legend2Text
```

We can see that each `Graphic` has its own `Text` attribute: `Legend1Text` and `Legend2Text`.
They both have a different `String`.
The leading $ character indicates that we won't display a raw text but that we'll use the content as a key for the localization system.
So in the end, the `Legend1` object will display the localized string with the key `Content`, and `Legend2` the one which has the key `Lang`.

Everytime we will switch to another language, both `orxTEXT` objects (ie. `Legend1Text` and `Legend2Text`) will have their content updated automagically in the new selected language.
As we saw earlier, we can catch the `orxLOCALE_EVENT_SELECT_LANGUAGE` event to do our own specific processing in addition, if needed.

We can also see that `Legend1Text` is using a font defined in the language section with the key `LocalizedFont`. This way the font used depends on the current language. If none is defined, it'll revert to orx's default one. This come in handy when you want separate fonts for different languages

using different alphabets.
In our case, one language out of two is defining `LocalizedFont` to be `CustomFont` and the `Chinese` language defines it to `CustomChineseFont`.

Let's now see how custom fonts are declared in orx.

```
[CustomFont]
Texture = penguinattack.png
CharacterList = " !""#$%&'()*+,-
./0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmnopqrstuvwxy
z{|}~€,ƒ„…†‡ˆ‰Š‹ŒŽ‘’“”•–—˜™š›œžŸ ¡¢£¤¥¦§¨©ª«¬-
®¯°±²³´µ¶·¸¹º»¼½¾¿ÀÁÂÃÄÅÆÇÈÉÊËÌÍÎÏÐÑÒÓÔÕÖ×ØÙÚÛÜÝÞßàáâãäåæçèéêëìíîïðñòóôõö÷øù
úûüýþÿ"
CharacterSize = (19, 24, 0)

[CustomChineseFont]
Texture = customchinesefont.png
CharacterList = "Orx志是标的这"
CharacterSize = (24, 24, 0)
CharacterSpacing = (2, 2, 0)
```

The first line specifies the `Texture` that contains our font. Nothing really new here.

The second line, however, is a bit special. It contains all the characters defined in our font texture, in order of appearance.
Note that we have to double the " character inside a config block value so as to get the actual " character as part of the string.
Here we define all the characters (UTF-8/ANSI).

Lastly, the `CharacterSize` property defines the size of a single character.

The Chinese font was automatically generated by a tool called orxFontGen, using a TrueType font called `fireflysung.ttf`, and only contains the characters we need for our texts.
As we only need very few characters here, the result is a micro-font.
orxFontGen also defines a property called `CharacterSpacing` that matches empty spaces in the texture.
Empty spaces are useful when displaying anti-aliased text to prevent artefacts from neighboring characters to appear on the edges.

Note: As you can see, custom fonts need to be monospaced, with all the characters assembled in a grid manner, without any extra spacing.

Resources

Source code: 10_Locale.cpp

Config file: 10_Locale.ini

1) not to be mistaken with orx's internal interactive console

[2] which are implemented in `orx.h`

[3] in this case the preprocessor macro

```
__orxCPP__
```

will be automatically defined

[4] the ones given as parameter don't contain the executable name which is needed to determine the main config file name

[5] which uses WinMain() instead of main()