本页由 六月的流光 翻译自 官方的教程

# 独立程序与本地化教程（stand alone & localization）

## 综述

这是我们的第一个C++基础教程。它也展示了如何使用orx编写独立的可执行文件和使用本地化模块（orxLOCALE）。

由于我们在此教程中不再使用默认的可执行文件，它的代码会被直接编译成可执行文件而不是外部库。

这暗示了我们不再有前几个教程中的如下默认行为：[1)]

- F11 切换垂直同步
- Escape 退出
- F12 截屏
- 退格键(Backspace) 重新载入全部配置文件
- 配置文件中的[Main] 配置段被用于加载一个插件（“GameFile” 键）

一个直接基于orx的程序[2)]，默认收到orxSYSTEM_EVENT_CLOSE事件后不退出。为了改变这种情况，我们必须使用orx_Execute() 函数（如下link）或者自己处理。

查看之前关于basic object creation, clock handling, frames hierarchy, animations, cameras & viewports, sounds & musics, FXs, physics and scrolling 的基础教程以获得更多的信息。

我们现在只能靠自己了，所以必须自己写main 函数和手动初始化orx（好处是如果需要，我们可以任意指定要使用哪些模块，停用显示模块或其他模块。如果我们还是想要半自动初始化orx（我们可以使用orx_Execute() 函数。

本教程将通过这个辅助函数使用orx（但是如果它的行为不符合你的需求也可以不使用。

这个辅助函数将会确保所有部分正确初始化和妥善退出。
它也将确保clock模块正常运作（作为 orx的核心部分）
orxSYSTEM_EVENT_CLOSE事件发送时退出。
此事件在关闭窗口时发送，但是也可以按你的标准发送（例如按下Esc键）。

这些代码同样也是一个基础的C++示例，用来演示如何通过C语言以外的语言来使用 orx（
本教程本来应该用另外一种更好的方式组织起来（例如分割成多个头文件）但是我们希望每个基础教程都只有一个文件。
这个独立可执行文件会创建一个终端（和默认的orx可执行文件一样），但是如果你喜欢也可以创建一个没有终端的可执行文件。
为了实现上述目标，你只需要提供一个包含该可执行文件名的argc/argv风格的参数表即可。
如果不这样的话，默认加载的配置文件是 orx.ini（取代基于可执行文件名的配置文件（例如10_StandAlone.ini）。

对visual studio 用户（windows）而言，可以轻松通过撰写WinMain() 替代main() 函数，并且获取可执行文件的名称（或者硬编码实现，正如本教程中可耻地这么做鸟^.^）。

本教程简单的现实了orx的logo和一个本地化的说明。按空格键或者点击鼠标左键以循环显示所有可用语言的说明文本。

下面是一些你可以在本教程中找到的核心元素的解释：

- 运行函数（Run function）：不要在这里放置任何逻辑代码，它只是一个处理默认核心行为（例如跟踪退出或者更改locale）或简要描述一些东西的主干。由于它直接从main循环中调用并且不是clock系统的的一部分，时间一致性无法被强制执行。对你所有的主游戏可执行文件，请在此创建（或者使用已有的）clock并且注册你的回调函数。
- 事件处理器（Event handlers）：当一个事件响应函数返回orxSTATUS_SUCCESS后，对此事件就不会再调用其他事件响应函数。另一方面，如果返回的是orxSTATUS_FAILURE，事件会传递给其他正在监听这个事件的响应函数。我们将监视locale事情以便当选择的语言切换时更新我们的说明文本。
- orx_Execute()：通过我们自定义 的函数（Init，Run和Exit）初始化和执行orx。当然，如果它不符合我们的需求，我们可以不使用这个辅助函数并且手动处理所有事情。你可以通过参考orx_Execute()的内容[3]了解怎么这样做。

# 详细说明

让我们从包含的文件开始。

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

这就是你要使用orx所需要包含的唯一文件。它在C和C++[4]编译器下都工作良好。

现在看下我们的StandAlone类，包含orx Init()，Run()和Exit()回调函数。

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

  void SelectNextLanguage();

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

private:
  orxSTATUS                     InitGame();

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

所有的回调函数实际上都可以定义在任何类之外。这里这么做只是演示当你需要的时候你可以这么做。我们看到StandAlone类也包含了我们的logo对象和一个当前选中的语言的索引。

现在看一下Logo类的定义：

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

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

这里没有什么特别的，我们用指针指向一个orxOBJECT作为我们的logo另一个用来指向显示的本地化说明。如你所见，我们在这个可执行文件里将不会使用这个可执行文件的所有引用，我们只是保持它们以便显示在销毁Logo对象时被正确地清理。如果我们不手动做，在退出的时候orx会替我们搞定。

现在让我们看一下它的构造函数：

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

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

用前面的教程中讲的方法，我们创建了两个对象Logo和 Legend并且我们通过 orxObject_SetUserData() 把Logo C++对象链接到它对应的orx对象上。

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

这里只是简单的删除了我们的两个对象。

现在看看我们的main函数：

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

  return EXIT_SUCCESS;
}
```

正如我们看见的，我们使用 orx_Execute() 辅助函数来初始化和执行orx 这样我们需要提供我们的可执行文件名称和命令行参数和三个回调函数Init() Run() 和Exit()
只有当orx退出时我们才从这个辅助函数退出。 我们快速浏览一下windows的无命令行版本。

```
#ifdef __orxMSVC__
```

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

  // Done!
  return EXIT_SUCCESS;
}

#endif
```

和旧的main() 版本一样除了我们使用orx_WinExecute()辅助函数来计算正确的命令行参数并使用它[5]。 这只能在一个无命令行（终端）的windows 游戏下工作。[6]

现在看看我们的Init() 代码是怎么样的：

```
orxSTATUS StandAlone::Init()
{
  orxLOG("10_StandAlone Init() called!");

  return soMyStandAloneGame.InitGame();
}
```

我们简单地用StandAlone实例中的 InitGame()方法来初始化。

让我们看看它的内容：

```
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");
```

我们简单地注册了一个捕捉所有orxEVENT_TYPE_LOCALE事件的回调函数。
然后实例化所有在配置文件中定义的可用语言。我们通常可以用orxLOG()宏来记录（在屏幕上和文件里），但是我们这里用C++的方式来实现以表示多样性。
我们通过创建一个视口来结束，就像我们在之前所有教程中做的那样。

现在看下我们的Exit()回调函数：

```
void StandAlone::Exit()
{
  delete soMyStandAloneGame.m_poLogo;
```

```
  soMyStandAloneGame.m_poLogo = NULL;

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

没有什么特别的，简单地在这里销毁了Logo对象。

让我们看下Run()回调函数：

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

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

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

  return eResult;
}
```

这里完成了两件事情。
首先我们当CycleLanguage被激活时，我们切换到下一个可用的语言，其次如果Quit被激活，我们简单地返回orxSTATUS_FAILURE。
当Run()回调函数返回orxSTATUS_FAILURE时（orx使用orx_Execute()辅助函数）将会退出。

让我们快速的浏览一下 SelectNextLanguage() 方法。

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

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

基本上我们只是移动到下一个可用的语言（如果到最后一个则循环到列表的开头）并通过orxLocale_SelectLanguage() 函数选择它。
当我们这么做时，如果使用一个本地化字符串的已创建orxTEXT对象将会被自动更新。我们会在下面的配置描述中看到如何实现。
我们可以在EventHandler回调函数中做捕捉任意语言的选择事件。

```
orxSTATUS orxFASTCALL StandAlone::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;
}
```

如你所见，我们只跟踪orxLOCALE_EVENT_SELECT_LANGUAGE 事件来显示新选择的语言。

现在我们完成了本教程代码的部分。让我们看看配置吧。

首先，正如你已经见到的，我们使用对不同的平台使用不同的文件夹。
换而言之，Mac OS X的教程放在 /mac文件夹，Linux的教程放在/linux，以此类推。默认的情况下，orx会查找当前目录下的主配置文件。
为了避免在不同的平台下都有重复的配置文件，我们创建了一个简单地配置文件包含了父文件夹中配置文件的全部信息。

```
@../10_StandAlone.ini@
```

所有的内容如上。就像你在 配置文件语法说明中看到的，我们可以通过@path/to/FileToInclude@ 在一个配置文件中包含其他的配置文件。[7)]

现在我们看看父文件夹中存储的配置文件(ie. ../10_StandAlone.ini).

首先定义我们的display配置段：

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

如你所见，我们将要创建一个分辨率为800×600的窗口，并且定义了他的标题。

现在我们要提供视口（viewport）和摄像头（camera）配置段的信息：

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

[Camera]
FrustumWidth    = @Display.ScreenWidth
```

```
FrustumHeight = @Display.ScreenHeight
FrustumFar    = 2.0
Position      = (0.0, 0.0, -1.0)
```

与我们在 视口教程中提到的没有任何区别。

现在看看输入□inputs□是怎么定义的

```
[Input]
SetList = MainInput

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

在Input配置段，我们定义我们所有的输入集合。本教程中我们只会用到一个叫做MainInput的集合，但我们也定义其他任意想要使用的集合（例如，主菜单一个，游戏中一个，等等□□MainInput集合包括3个映射：

- KEY_ESCAPE 会触发名为Quit的输入
- KEY_SPACE 和 MOUSE_LEFT 都会触发名为CycleLanguage的输入

我们可以在这个配置段加入任意多的输入集合并且将它们绑定到按键、鼠标按钮（包括滚轮上/下）、游戏摇杆按钮甚至游戏摇杆的方向轴。
现在让我们看看怎么定义将要被 orxLOCALE模块使用的语言。

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

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

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

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

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

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

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

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

为了定义本地化需要的语言我们要定义一个Locale 配置段和一个包含我们所需要全部的语言的列表。
然后我们需要为每个语言定义一个配置段，并且包含我们需要的每一个内容(在这里是“Content”“和” Lang”)我们将它们设成本地化的文字。
用同样的方法，我们为每两个语言中的一个定义了LocalizedFont，而且我们将使用它来指定一个与本地化文字/语言组合在一起的特殊字体。

由于本地化系统是基于orx配置部分，我们可以使用它的继承能力来简化把语言加入列表的过程（例如在另一个外部文件中），甚至也可以完善曾经只被部分定义的语言。

现在看看我们是怎么定义Logo对象：

```
[LogoGraphic]
Texture = ../../data/object/orx.png
Pivot   = center

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

又一次，所有的内容我们都已经在对象教程object tutorial中涵盖了。

如果你对我们定义了哪些FX感到好奇，你可以直接查看10_StandAlone.ini ，但是我们不会在这里讨论它们的细节。

下一个要查看的是：我们的Legend对象：

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

奇怪吧！其实它只是一个我们由两子对象繁衍出来的空对象。 

通过这样的编码方式，我们创建了一个叫做Legend 的单独对象但显然最终将超过一个对象。同样的技术也可以用来创建一组的对象，或者是完成一个场景，而不是一个一个创建。同样可以将对象通过 ChildList 串联起来并只在我们的代码中创建一个单独的对象而同时有数百个对象被创建。

然而我们对它们不会有直接的指针，这意味着我们将不可能直接操作它们。

话虽这么说，对所有非交互/后台对象这却不是一个问题。

同时请注意他们的帧（参考：frame tutorial）会影响ChildList ‘串联’ 的继承。

好了，现在让我们回到Legend1 和Legend2两个对象。

```
[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
```

它们看起来很基本，都是用了相同的 FX□ColorCyle2□□它们也都有一个位置并且各自有它们的Graphic□

注意：我们也可以看到我们为它们定义了ParentCamera属性。这意味着最终它们实际的父对象为Camera而不是Legend对象。

然而Legend还将是它们的拥有者，这说明它们将会在Legend被销毁时自动被销毁。

现在让我们看一下它们的Graphic对象作为结束。

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

[Legend2Text]
String = $Lang

[Legend1Graphic]
Pivot = center
Text  = Legend1Text

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

我们可以看到每一个Graphic都有自己的Text 属性□Legend1Text和Legend2Text□ 它们都有不同的String字段。

开头的$字符说明我们不会显示一个原始的文本但内容作为我们将会把内容作为本地化系统的关键字。

所以在最后□Legend1对象会显示该关键字Content的本地化字符串□Legend2会显示关键字Lang的本地化字符串。

每一次我们会切换到另一个语言，所有的orxTEXT对象（例如Legend1Text和 Legend2Text□会根据新选择的语言自动更新它们的内容。：）

正如我们早前看到的，如果需要，我们还可以捕获 orxLOCALE_EVENT_SELECT_LANGUAGE事件来进行特定的处理。

我们也可以看到，在配置段Legend1Text中用LocalizedFont指定了一个字体。这种情况下，字体的使用就依赖于目前选择的语言。假如没有字体被指定，那么它将使用Orx的默认字体。
当你想要将使用不同字符的语言的字体分开时，这将非常有用。
在现在例子里，二分之一的语言就使用了LocalizedFont配置定义字体为CustomFont。

让我们现在来看下怎么在orx中声明一个自定义字体吧。

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

第一行指定了包含我们字体的Texture□没有什么新的内容。
二行，却有一点特别。为了显示，它包含了定义在我们字体纹理中的所有字符。
注意到我们在配置块值里面把 “ 写了两次以便得到真正的 ” 字符作为字符串的一部分。
这里我们定义了ISO Latin 1中所有的字符。
最后□CharacterSize属性定义了每一个字符的大小。

注意：如你所见，要在一个格子管理器中组装所有的字符而没有多余的空间，要求自定义字体等宽（即等宽字体）。
另外□iarwain实现了一个支持unicode的字体图片生成工具， 具体见这里□

# 资源

源代码：10_StandAlone.cpp

配置文件：10_StandAlone.ini

1)
译注：内置于默认可执行文件，同时下面的几点原本也都是否定句式，但与这里的否定矛盾，原意应该是下面这些特性会在StandAlone版本中失效
2)
即不再依靠ORX 启动器
3)
在orx.h中实现
4)
在这种情 况下预编译宏

```
__orxCPP__
```

会被自动定义
5)
这些给出的参数并没有包含我们需要的用来决定主配置文件名的可执行文件的名字
6)
使用WinMain()替代main()
7)
译注：这里怀疑是作者的问题,template files实际是想链接到WIKI的配置的说明上去

From:
https://www.orx-project.org/wiki/ - **Orx Learning**

Permanent link:
**https://www.orx-project.org/wiki/cn/orx/tutorials/standalone?rev=1518583598**

Last update: **2025/09/30 17:26 (14 months ago)**