Tutorial de Ejecutable Independiente

Sumario

Este es nuestro primer tutorial básico de C++. También muestra como escribir un ejecutable independiente usando orx y como usar el módulo de localización (`orxLOCALE`).

Como **NO** estamos usando el ejecutable por defecto para este tutoriales, su código será directamente compilado en un ejecutable y no dentro de una librería externa.

Esto implica que **NO** tendremos comportamiento codificado por defecto que tuvimos en los tutoriales anteriores:

- F11 no afectará el cambiador de sincronia vertical.
- Escape no saldrá de la aplicación automáticamente.
- F12 no captura una imagen
- Backspace no recarga ficheros de configuración
- La sección [Main] en el fichero de configuración no será usada para cargar un plugin (`GameFile` llave)

Un programa basado directamente en orx [1)], por defecto, **NO** saldrá de la aplicación si recibe el evento orxSYSTEM_EVENT_CLOSE.
Para hacer esto, o tendriamos que usar la función auxiliar `orx_Execute()` (ver debajo) o manejarlo por nuestra cuenta.

Ver los anteriores tutoriales básicos para más información sobre la creación básica de objetos, manejo del reloj, jerarquía de fotogramas, animaciones, cámaras & vistas, música & sonido, efectos(FXs), física y desplazamiento.

Como estamos por nuestra cuenta aquí, necesitamos escribir la función principal e inicializarla manualmente con orx.
Lo parte buena es que podemos entonces especificar que módulo queremos usar, y desactivar la pantalla o cualquier otro módulo a voluntad, si fuera necesario.

Si quisieramos mantener una semi automática inicialización de orx, podemos usar la función `orx_Execute()`.
Este tutorial cubrirá el uso de orx con su función auxiliar, pero puedes decidir si no la usas su su comportamiento no sirve para tus necesidades.

Esta función auxiliar tendrá cuidado de inicializar todo correctamente y salir adecuadamente.
Estará también segura que el módulo del reloj está marcando constantemente (como parte del núcleo de orx) y que podamos salir si el evento orxSYSTEM_EVENT_CLOSE fue enviado.
Este evento es enviado cuando cerramos una ventana, por ejemplo, pero puede ser enviado por criterio propio (la tecla escape es presionado, por ejemplo).

Este código es un ejemplo básico de C++ para mostrar como usar orx sin tener que escribir código de C.
Este tutorial pudo haber estado mejor estructurado de una mejor manera (cortandolo en piezas con encabezados de ficheros, por ejemplo) pero queremos mantener un solo fichero por tutorial *básico*.

Este ejecutable independiente también crea una consola (como hace el ejecutable de orx por defecto), pero tu puedes tener tu propio programa sin consola si así lo deseas.
A fin de lograr eso, solo necesitas proveer un listado de argumentos que contenga el nombre del ejecutable.
Si no, el fichero cargado por defecto será orx.ini en vez del que está basado en el nombre de nuestro ejecutable (ej. 10_StandAlone.ini).

Los usuarios(windows) de Visual Studio, fácilmente pueden lograr esto escribiendo una función `WinMain()` en vez de `main()`, y obteniendo el nombre del ejecutable (o hacerlo a mano, como se hace sin pudor en este tutorial 😀)

Este tutorial simplemente muestra el logo de orx y una leyenda localizada. Presione espacio o el botón click izquierdo para pasar por todas las lenguas disponibles para la leyenda del texto.

Algunas explicaciones acerca de elementos del núcleo puedes encontrarlas en este tutorial:

- `Función correr`(`Run function`): No ponga *ningún* código lógico aquí, es solo en la columna vertebral donde puedes manejar por defecto los comportamientos del núcleo(rastreando la salida o cambiando la localización, poer ejemplo) o perfilar algunas cosas. Como esto es llamado llamado directamente desde un ciclio principal y no como parte del reloj del sistema, la consistencia en el tiempo no se puede imponer. Para todas las ejecuciones principales de tu juego, por favor crea un reloj(o usa uno existente) y registra tu llamada de retorno(callback) a el.
- `Controladores de Evetos`(`EventHandlers`): Cuando un controlador de evento retorna orxSTATUS_SUCCESS, ningún otro controlador será llamado después de el por el mismo evento. En la otra mano, si orxSTATUS_FAILURE, es retornado, el procesamiento de eventos continuará durante ese evento si los controladores de otros están escuchando este tipo de evento. Vamos a supervisar los eventos de localización para actualizar el texto de nuestra leyenda, cuando el idioma seleccionado se cambia.
- `orx_Execute()`: Inicia y ejecuta orx usando nuestra propia función definida(Init, Run y Exit). Podemos, claro, no usar este auxiliar y controlar todo manualmente, si este comportamiento no suple nuestras necesidades. Puedes echar un vistazo al contenido de `orx_Execute()` [2] para tener una mejor idea en como se hace esto.

Detalles

Empecemos con los includes.

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

Eso es todo lo que necesitas para incluir a fin de utilizar orx. Este include trabaja igualmente con un compilador de C o C++ [3].

Veamos ahora a nuestra clase `Independiente`(`StandAlone`) que contiene las llamadas de retorno de orx, `Init()`, `Run()`, `Exit()`.

```
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;
};
```

Todas las llamadas de retorno pueden actualmente haber sido definidas fuera de cualquier clase. Esto es hecho justamente aquí para mostrar como hacerlo si lo necesitaras luego.
Podemos ver que nuestra clase `StandAlone` también contiene nuestro objeto `Logo` y un índice para el correspondiente idioma seleccionado.

Echemos un vistazo a la definición de nuestra clase `Logo`.

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

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

Nada fantasioso aquí, tenemos una referencia a un `orxOBJECT` que será nuestro logo y otro más que será para mostrar la leyenda localizada.
Como puedes ver no usamos una referencia a todo en este ejecutable, que acabamos de mantenerlos, a fin de mostrar una correcta limpieza cuando nuestro objeto `Logo` es destruido. Si no queremos hacerlo manualmente, orx tendrá cuidado de eso de todas formas cuando cerremos.

Veamos ahora su constructor.

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

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

Como hemos visto en tutoriales anteriores creamos nuestros dos objetos (Logo y Leyenda) y enlazamos nuestro Objeto C++ Logo a su equivalente en orx usando `orxObject_SetUserData()`.

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

Fácil de limpiar aquí, ya que solo elimina los dos objetos.

Veamos ahora nuestra función principal.

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

  return EXIT_SUCCESS;
}
```

Como podemos ver, estamos usando el auxiliar `orx_Execute()` que inicializará y ejecutará orx por nosotros.

Con el fin de hacer esto, necesitamos proveerle el nombre de nuestro ejecutable y los parámetros de línea de comando, junto con tres devoluciones de llamada de: `Init()`, `Run()` y `Exit()`.
Solo saldremos de esta función auxiliar cuando orx termine.

Tengamos un pequeño vistazo a la versión de consola para windows.

```
#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
```

Lo mismo que para la tradicional version `main()` excepto que usamos el auxiliar `orx_WinExecute` que calculará los parámetros correctos en la línea de comandos y los usará. [4)]

Esto solo funciona para un juego en windows sin consola [5)].

Veamos ahora como luce nuestro código de `Init()`.

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

  return soMyStandAloneGame.InitGame();
}
```

Simplemente inicializaremos nuestra instancia `StandAlone` llamando a su método `InitGame()`.
Veamos su contenido.

```
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");
```

Simplemente registramos un callback para capturar todos los eventos `orxEVENT_TYPE_LOCALE`.
Instanciamos entonces nuestro objeto `Logo` que contiene ambos, logo y leyenda.
También se emiten todos los idiomas disponibles que han sido definidos en los ficheros de configuración.
Podemos tener usando la macro `orxLOG()` para registrar como es usual (en pantalla y en fichero), pero lo hacemos a la manera de C++ aquí para mostrar cierta diversidad.
Terminamos por crear nuestra vista, como se ha dicho en todos los tutoriales anteriores.

Veamos ahora nuestra callback `Exit()`.

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

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

Simple borrado del objeto `Logo` aquí, nada sorprendente.

Veamos ahora nuestra callback `Run()`.

```
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;
}
```

Se hacen dos cosas aquí.
Primero cuando la entrada `CycleLanguage` es activada cambiamos para el siguiente idioma disponible, entonces cuando `Cerrar(Quit)` es activado, simplemente retornamos `orxSTATUS_FAILURE`.
Cuando la callback `Run()` retorna `orxSTATUS_FAILURE` orx (cuando es usado con el auxiliar `orx_Execute()`) cerrará.

Veamos rápidamente al método `SeleccionaPróximoIdioma(SelectNextLanguage)`.

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

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

Básicamente vamos al próximo idioma disponible (regresando al principio de la lista cuando llegamos al último) y lo seleccionamos con la función `orxLocale_SelectLanguage()`.
Cuando hacemos esto, todos los objetos `orxTEXT` creados se actualizarán automáticamente si ellos usan una cadena localizada. Veremos como se hace debajo en la descripción de la configuración.
Podemos atrapar la selección de cualquier idioma como se hace en nuestra `EventHandler` callback.

```
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;
}
```

Como puedes ver, solo rastreamos el evento `orxLOCALE_EVENT_SELECT_LANGUAGE` aquí, así como para mostrar que es el nuevo idioma seleccionado.

Hemos terminado ahora con la parte del código de este tutorial. Veamos la configuración.

Primero que todo, como has podido ver, usamos diferentes carpetas para diferentes arquitecturas. En otras palabras, el tutorial para Mac OS X está en la carpeta /mac, la de Linux en /linux, etc...

Por defecto, para un proyecto independiente, orx mirará en la carpeta correspondiente (ej. la carpeta que contiene el .exe) para encontrar el fichero de configuración principal.

Como no queremos duplicar el fichero de configuración en las carpetas de todas las arquitecturas, creamos uno muy simple con el único propósito de incluir al que contiene toda la información y que es la carpeta padre.

Veamos ahora como hacemos esto mirando en el contenido de `10_StandAlone.ini` de unas de las subcarpetas (ej. una que es guardada en la misma carpeta que el ejecutable del tutorial).

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

Es todo lo que podemos encontrar ahí. Como puedes ver en los ficheros de plantillas, podemos incluir otros ficheros de configuración escribiendo `@path/to/FileToInclude@`.

Miremos en el fichero de configuración quien es guardado en la carpeta padre (ie. ../10_StandAlone.ini).

Definamos nuestra pantalla.

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

Como puedes ver, estamos creando una ventana de resolución 800×600 y definiendo su título.

Necesitamos ahora proveer información para nuestras vista y cámara.

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

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

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

Nada nuevo aquí, ya que todo ya estaba cubierto en el tutorial de vistas.

Veamos que entradas son definidas.

```
[Input]
SetList = MainInput

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

En la sección `Entrada(Input)`, definimos todos nuestras entradas. En este tutorial solo usamos una llamada `EntradaPrincipal(MainInput)` pero podemos definirla como muchas otras que queramos (por ejemplo, una para el menú principal, una para el juego, etc...).

La `MainInput` contiene 3 teclas:

- `TECLA_ESCAPE(KEY_ESCAPE)` disparará la entrada llamada `Quit`
- `TECLA_ESPACIO(KEY_SPACE)` y `RATON_IZQUIERDO(MOUSE_LEFT)` que ambos dispararán la entrada llamada `CycleLanguage`

Podemos añadir tantas entradas como queramos en esta sección y atadlas a las teclas, botones del ratón(incluyendo rueda arriba/abajo), botones del joystick o ejes del joystick.

Veamos como definimos idiomas que serán usados por el módulo `orxLOCALE`.

```
[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
```

Para definir idiomas para localización solo necesitamos definir una sección `Locale` y definir una `ListaIdiomas`(`LanguageList`) que contendrá todos los idiomas que necesitamos.
Después de esto necesitamos definir una sección por idioma y para cada tecla necesitada (aquí `Content` y `Lang`) hemos puesto su texto localizado.
De la misma manera, definimos `FuenteLocalizada`(`LocalizedFont`) para un idioma o dos, y la usaremos para especificar una fuente determinada basada en la combinación texto/idioma.

Como el sistema de localización está basado en una configuración de orx, podemos usar su capacidad hereditaria para fácilmente añadir nuevos idiomas a la lista(en otro fichero externo, por ejemplo), incluso para completar idiomas que han sido parcialmente definidos.

Veamos ahora como definimos nuestro objeto `Logo`.

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

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

De nuevo, todo lo que podemos ver aquí está cubierto en el tutorial de objeto.
Si eres curioso puedes mirar directamente en 10_StandAlone.ini para ver que tipos de FXs definimos, pero no los detallaremos aquí.

Próxima cosa a chequear: nuestro objeto `Leyenda`(`Legend`).

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

Sorpresa! Actualmente es un objeto vacío que reproducirá dos objetos hijos: Legend1 y Legend2.


El código-sabio fue creado en un solo objeto llamado Legend pero aparentemente hemos terminado con más de un objeto.
El mismo tipo de técnica puede ser usada para generar un sin número grupo de objetos, o un completo escenario, por ejemplo, sin tener que crearlos uno a uno con código-sabio.
Es posible encadenar objetos con `ChildList` y únicamente crear un solo objeto en nuestro código y tener miles de objetos actuales creados.
Sin embargo, no tenemos punteros directos en ellos, lo que significa que no seremos capaces de manipularlos directamente.
Siendo esto dicho, para todos los objetos no-interactivos/fondos esto usualmente no es un problema.
Sea consciente de que sus fotogramas (rf. tutorial de fotogramas) se reflejarán en la jerarquía de la cadena `ChildList`.

Ok, ahora regresemos a nuestros dos objetos, Legend1 y 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
```

Eso luce muy básico, ellos dos están usando el mismo FX(`ColorCycle2`), ambos tienen una posición y cada uno tiene su propio `Graphic`.

PD: Podemos ver que definimos el atributo CámaraPadre(`ParentCamera`) para ambos. Esto significa que su actual padre será la cámara y no el objeto Legend finalmente.
Sin embargo Legend seguirá siendo su propietario, lo quiere decir que ellos serán automáticamente borrados cuando Legendd sea eliminado.

Terminemos de echarle un vistazo a sus objetos `Graphic`.

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

[Legend2Text]
String = $Lang

[Legend1Graphic]
Pivot = center
Text  = Legend1Text
```

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

Podemos ver que cada Graphic tiene su propio atributo Text: Legend1Text y Legend2Text. Ellos ambos tienen diferentes Cadenas(String). El carácter inicial $ indica que no mostramos un texto crudo pero usamos el contenido como llave para el sistema de localización.
Entonces, al final, el objeto Legend1 mostrará la cadena localizada con la llave Content, y Legend2 la única que tiene la llave Lang.

Todo el tiempo cambiaremos a otro idioma, ambos objetos orxTEXT(ej. Legend1Text y Legend2Text), tendrán su propio contenido automáticamente actualizado en la nueva selección del idioma. 😀
Como vimos anteriormente, podemos capturar el evento orxLOCALE_EVENT_SELECT_LANGUAGE para hacer nuestro propio proceso de adición, si lo necesitáramos.

Podemos ver también que Legend1Text está usando una fuente definida en la sección del idioma con la llave LocalizedFont. De esta manera las fuentes usadas dependen del correspondiente idioma. Si no es definido, será revertido a una forma predeterminada de orx. Esto es muy útil cuando quieres separar fuentes para idiomas diferentes usando alfabetos diferentes.
En nuestro caso, uno de los idiomas está definiendo FuenteLocalizada(LocalizedFont) para ser FuentePersonalizada(CustomFont) y el idioma Chino(Chinese) lo define FuenteChinaPersonalizada(CustomChineseFont).

Veamos ahora como las fuentes personalizadas son declaradas en orx.

```
[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)

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

La primera linea especifica la Texture que contiene nuestra fuente. Nada nuevo hasta ahora.

La segunda linea, sin embargo, es un poco especial. Contiene todos los caracteres definidos en la textura de nuestra fuente, en orden de aparición.
Tenga en cuenta que tenemos que duplicar el caracter dentro de un bloque de configuración de valor a fin de obtener el verdadero caracter como parte de la cadena.
Aquí definimos todos los caracteres de ISO Latin 1.

Por último, la propiedad `TamañoCaracter(CharacterSize)` define el tamaño de un caracter simple.

La fuente Chinese fue automáticamente generada por una herramienta llamada orxFontGen, usando una fuente TrueType llamada `fireflysung.ttf`, y solo contiene los caracteres que necesitamos para nuestros textos.
Como solo necesitamos muy pocos caracteres aquí. El resultado es una micro-fuente.
orxFontGen define también una propiedad llamada `EspaciadoDeCaracter(CharacterSpacing)` que coincide con los espacios vacíos en la textura.
Los espacios vacíos son útiles cuando se muestra el texto suavizado para evitar artefactos de caracteres vecinos que aparezcan en los bordes.

Nota: Como has podido ver, las fuentes comunes necesitan ser monoespaciadas, con todos los caracteres juntos en forma de malla, sin ningún espaciado extra.

Recursos

Código fuente: 10_StandAlone.cpp

Fichero de configuración: 10_StandAlone.ini

1)
ej. sin la ayuda del lanzador orx
2)
que está implementado en `orx.h`
3)
en este caso el macro preprocesador

```
__orxCPP__
```

será automáticamente definido
4)
lo pasado como parámetro no contiene el nombre del ejecutable que se necesita para determinar el nombre del fichero de configuración principal
5)
que usa WinMain() en vez de main ()