# Fabriquer un PlugIn RealBasic

Avérous Julien-Pierre

28 décembre 2004

Merci à Sébastien Gallerand pour la correction orthographique.

# Table des matières

# 1 Introduction

Ce document va vous permettre de vous apprendre les bases de l'utilisation du SDK de RealSoftware qui permet de réaliser des Plug-Ins pour RealBasic. Il n'est ni là pour vous apprendre le C/C++ qui doit être chose connus pour lire ce document, ni là pour refaire toute la documentation du SDK. C'est pourquoi il traite de manière rapide le sujet grâce à des exemples les plus standards. Ce sera à vous ensuite de rechercher dans la documentation du SDK tel ou tel élément bien particulier.

# 2 Configuration

## 2.1 CodeWarrior 9

### 2.1.1 Installation du SDK

Il faut tout d'abord télécharger le SDK sur le site de RealBasic. Une fois fait, décompressez l'archive, et copiez le dossier dans le dossier où se trouve `CodeWarrior IDE`. Renommez le dossier `RealBasic SDK`. Ce n'est pas une obligation, mais ça permettra de bien savoir de qu'elle dossier on parle.

### 2.1.2 Configuration du projet

Créer un nouveau projet de type `MacOS C++ Stationery`. Validez puis choisissez `Mac OS Carbon > Mac OS Toolbox > C++ Toolbox Carbon`. Une fois le projet chargé, créez un groupe `RealBasic Plug SDK`. Glissez dans ce groupe le fichier `PluginMain.cpp` qui se trouve dans `RealBasic SDK/Glue Code`. Passez sur le target `Carbon Toolbox Final`, et supprimez le groupe `Resource`. On va maintenant configurer le target.

**Access Paths - User Path**

Sont normalement déjà présents `{Project}` et `{Compiler}RealBasic SDK/Glue Code` (qui s'est rajouté lorsque vous avez glissé le fichier `.cpp`). On va ajouter les includes du SDK. Cliquez dans la liste pour l'activer, cliquez sur `Add...`. Sélectionnez le dossier `RealBasic SDK/Includes`, mettez `Relative to` sur `CodeWarrior`

**PPC Target**

Mettez `Project Type` sur `Code Resource`, renommez FileName avec le nom voulus pour votre fichier Plug-In. Dans `Creator` mettez `RBv2`, dans `Type` mettez `RBPl`, dans ResType mettez `PLCN`, et enfin pour `ResID` mettez `128`. Placez enfin `Header Type` sur `None`.

**Property List**

Je vous conseille de mettre `Property List Compiler Output` sur `Output as file` et `File Name` avec pour texte `Info.plist`

**C/C++ Language**

De même, rien d'obligatoire : je vous conseille de cocher uniquement `Enable bool Support`, `Enable whar_t Support`, `Bottom-up Inlining`, `Enable GCC Extensions`, `Use Unsigned Chars` et `Reuse Strings`

**C/C++ Preprocessor**

Remplacez `MacHeadersCarbon.h` par `RBCarbonHeaders.h`.

**PPC Linker**

Remplacez `__start` du champ `Main` par `main`.

## 2.2 Xcode

### 2.2.1 Installation du SDK

Il faut tout d'abord télécharger le SDK sur le site de RealBasic. Une fois fait, décompressez l'archive, et copiez le dossier `SDK/Examples/Xcode Mach-O Template/REALbasic Mach-O Plugin` (j'appellerais ensuite ce dossier "dossier Template") dans le dossier `/Library/Application Support/Apple/Developer Tools/Project Templates/`

### 2.2.2 Retouche du SDK

Ouvrez le dossier Template et ouvrez `REALbasic Mach-O Plugin.xcode`. Afficher les infos du Target. Allez dans `Build` puis effacez la valeur associée à la ligne `Info.plist File`. Allez ensuite à la ligne `Development Build Products Path` et entrez comme valeur «`PROJECTNAME`»`/Build Resources/Mac Carbon Mach-O/` et enfin à la ligne `Development Intermediate Path` mettez comme valeur

`build/`. Vous pouvez fermer le projet. Si un dossier Build se créait, supprimez le. Dans le dossier Template, créez cette structure de dossier : `/MachOPlugin/Build Resources/Mac Carbon Mach-O/`. Accédez ensuite au contenu du package du projet (click-droit puis afficher le contenu du paquet). Ouvrez le fichier `TemplateInfo.plist` avec un éditeur de .plist (par exemple Property List Editor d'Apple). Pour la clef `FilesToRename`, ajoutez un "Enfant" pour clef `MachOPlugin` et pour valeur «`PROJECTNAME`». Pour améliorer encore un peut votre dossier Template vous pouvez par exemple y copier le fichier `Read Me.txt` ou y mettre un alias (ou une copie) de l'application `RB Plugin Converter`[1].

### 2.2.3 Utilisation du SDK

Lancez Xcode, allez dans le menu `File > New Project`, sélectionnez `RealBasic Mach-O Plugin`. Entrez un nom de projet, un emplacement et validez.
Quand vous compilez avec Xcode, il crée un fichier `.dylib` (qui se trouvera dans le dernier dossier de la hiérarchie que vous avez créé dans le template). Il faut donc convertir le fichier (ou plutôt la hiérarchie de dossier. Voir la section suivante) en Plug-In RealBasic. C'est ce à quoi va nous servir `RB Plugin Converter`. Lancez l'application et cliquez sur `Convert Folder...` Sélectionnez le dossier où se trouve la hiérarchie qui mène à votre fichier `.dylib`. Il a le nom de votre projet, et se trouve dans votre dossier-projet. L'application vous demande alors où créer le Plug-In. Choisissez l'endroit que vous voulez, et enregistrez. Votre Plug-In est prêt à l'emploi.

## 2.3 RBv2 et RBX

Jusqu'à présent, le format des Plug-Ins était dit en RBv2. Chaque version du Plug-In était dans une ressource propre (Carbon dans PLCN, PPC dans PLPC, 68k dans PL68 et enfin x86 dans PL86). Avec CodeWarrior, on compile encore en RBv2 qui est parfaitement reconnu par RealBasic 5.5
Le format dit RBX n'utilise plus du tout les Resources Forks, mais les Data Forks. Effectivement, toutes les versions du Plug-In sont, avec ce nouveau format, écrite dans la zone de donnée du fichier.
Les Plug-Ins faits avec Xcode (les .dylib) sont convertis au format RBX grâce

[1] Cette application se trouve dans /SDK/Plugin Converter

à `RB Plugin Converter`.
Sachant qu'on ne peut pas mélanger directement les 2 formats (mettre par exemple du Carbon MachO RBX et du Carbon PEF RBv2) dans un même Plug-In, il faut obligatoirement créer un RBX si et seulement si une version du Plug-In est en RBX. Sinon du RBv2 est tout a fait acceptable. Pour cela il faut utiliser la hiérarchie de dossier qui est utilisé lors de la conversion d'un dossier en Plug-In. Par exemple la version Carbon MachO généré par Xcode doit se trouver dans le dossier `Mac Carbon Mach-O`, et la version Carbon PEF doit se trouver dans le dossier `Mac Carbon` au même niveau que le dossier MachO. La version Carbon PEF est un Plug-In standard RBv2 ou RBX généré à partir de CodeWarrior ou d'un autre compilateur ayant la possibilité de compiler ce genre de code.
Il est à noter que si vous décidez de faire un Plug-In Carbon, votre Plug-In n'est pas obligé de posséder une version PEF, mais dans ce cas, votre Plug-In ne pourra être utilisé qu'avec une application RealBasic MachO. Et enfin, il est important de dire que Xcode ne sait pas compiler des Plug-In Carbon PEF, d'où cette section explicative sur les 2 formats et sur l'obligation d'utiliser du RBX si une version du Plug-In est en RBX (format obligatoire pour utiliser les .dylib de Xcode).

# 3 Structure du code

Le code doit ressembler au minimum à ça :

```
#include "rb_plugin.h"

void PluginEntry() {

}
```

Le principe est assez simple : entre le `#include` et le `PluginEntry`, se trouvera le code en C/C++ et les définitions des fonctions, classes, contrôles utilisés par RealBasic. Ce que j'appelle définition, c'est la description textuelle de la fonction (description utilisée sous RealBasic pour par exemple la fenêtre Tips ou pour la vérification du code lors d'une compilation) et le lien créé entre le code en C/C++ et RealBasic.
Quand à PluginEntry, il permet de dire à RealBasic de charger les définitions de votre PlugIn. C'est essentiel pour que nos fonctions, classes, contrôles, etc. . .soient vus sous RealBasic.

## 3.1 Création d'une fonction

Les fonctions en C/C++ du Plug-In doivent être `static`. Ne me demandez pas pourquoi, ça doit être intrinsèque au fonctionnement de RealBasic. On peut par exemple déclarer le code suivant :

```
static int fois(int x, int y)
{
   return x*y ;
}
```

La fonction déclarée, on va maintenant écrire sa définition :

```
REALmethodDefinition foidDefn = {
   (REALproc)fois,REALnoImplementation,"Fois(x as integer, y as
integer) as integer"
} ;
```

Le premier élément est un pointeur sur votre fonction en C/C++, et le dernier élément est la définition formelle textuelle de votre fonction. Maintenant il faut dire à RealBasic de charger cette définition au démarrage. Ce qu'on fait par ceci :

```
void PluginEntry() {
   REALRegisterMethod(&foidDefn) ;
}
```

Vous pouvez maintenant compiler, et placer votre Plug-In dans le dossier `Plugins` de RealBasic (n'oubliez pas de convertir le dossier si vous travaillez avec Xcode). Une nouvelle fonction existe alors en global : `Fois(x,y)`

Une chose à savoir : pour les types non simples comme les Pictures ou les Strings, RealBasic a mis à votre disposition dans le SDK des structures en C/C++ qui permettent quand même de les utiliser dans votre code. Tous les types ne disposent pas (hélas) d'une structure d'accès en C/C++, ce qui fait que vous ne pouvez pas travailler avec tous les types de données de RealBasic. Je ne vais pas citer toutes les structures qui existent et encore moins les fonctions proposées par le SDK qui permettent de travailler dessus, car tout ceci est disponible dans le fichier d'aide `APIRef.html`. Je peux toutefois citer une structure : `REALstring`. Pour l'utiliser dans votre fonction en C/C++, faite comme n'importe quelle variable : `REALstring str`. Vous pouvez alors par

exemple utiliser la fonction `REALCString(str)` qui vous renvoie un pointeur de type `const char` qui va vous permettre d'accéder à la chaîne de caractères de RealBasic comme à une chaîne normale en C/C++.

## 3.2 Création d'une classe

Ça devient déjà moins simple que de déclarer de simples fonctions. Une classe RealBasic en C/C++ n'en est en fait pas une du tout. On utilise pour "émuler" votre classe des tableaux de définition qu'on lie avec une définition de classe. Pour les méthodes, on utilise un tableau de définition de méthodes, pour les variables on utilise un tableau de définitions de variables, etc...Les variables, qui n'ont pas été encore vues, ne doivent pas être placées n'importe où (par exemple en global) : elles doivent êtres placées dans une structure. Une fois que nous avons tous ces éléments, on doit faire la définition de la classe qui va permettre de dire si elle étend une autre classe, de dire quel est le tableau de définition des variables, le tableau de définition des méthodes ou encore le tableau de définition des événements, de dire quelles sont les fonctions en C/C++ qui vont être utilisées pour le constructeur et destructeur, quelle est la super-classe, etc...Bien sur, tous les champs de définitions de votre classe ne sont pas obligatoirement à renseigner. Vous pouvez très bien faire une classe possédant uniquement des méthodes ou uniquement des variables ou les deux. Le code de votre classe RealBasic doit commencer par la déclaration de la variable qui va contenir la définition de votre classe (a mettre juste après le `#include`, c'est le mieux). Par exemple `extern REALclassDefinition maClass;`.

### 3.2.1 Implémentation des méthodes

Comme pour la partie précédente, les méthodes sont des fonctions en C/C++. La différence avec une classe, c'est qu'il faut utiliser une liste de définition de méthode, et non pas enregistrer définition par définition. De plus, vos fonctions doivent avoir un paramètre de plus que d'habitude en premier paramètre. Il doit être de type `REALobject`. Ce paramètre contient en fait l'instance de votre classe, instance que RB fournit automatiquement à votre fonction lors de son appel via sa définition. Il ne faut donc pas rajouter cet argument à la description textuelle de votre fonction. Cette variable est par exemple utilisée pour accéder aux variables de votre classe RealBasic. Nous verrons ceci plus en détail dans le section sur l'implémentation des propriétés. Voici à quoi doit ressembler votre tableau de fonctions :

```
REALmethodDefinition classMethods[] = {
    { (REALProc) maFonction1, REALnoImplementation, "Foo1()" },
    { (REALProc) maFonction2, REALnoImplementation, "Foo2(x as
integer) as double" }
} ;
```

### 3.2.2 Implémentation des variables

Pour les variables, comme dit précédemment, il faut utiliser une structure qui les contiennent en dur en C/C++. Par exemple :

```
struct classData {
    int entier;
    bool booleen;
};
```

Puis définir nos variables à l'aide d'un tableau :

```
REALproperty classProperties[] = {
    { "Appearance", "Entier", "Integer", REALpropInvalidate,
    REALstandardGetter, REALstandardSetter,
FieldOffset(classeData, entier) },
    { "Behavior", "Booleen", "Boolean", REALpropInvalidate,
    REALstandardGetter, REALstandardSetter,
FieldOffset(classeData, entier) }
};
```

La première chaîne de caractères correspond à l'emplacement de la variable dans la fenêtre des propriétés. La deuxième, c'est le nom de la variable sous RealBasic, le quatrième et le cinquième élément permette de gérer automatiquement les variables (car sinon on peut passer par deux fonctions différentes : une quand on définit la valeur de la variable, et l'autre quand on la récupère. Ce point sera vu plus loin). Et enfin le dernier élément permet de dire dans quelle structure est la variable et sous quelle nom (Mis à 0 quand les variables ne sont pas gérées automatiquement).
Il est donc possible, comme dit précédemment, de gérer nous même l'accès aux variables. Pour cela, on doit définir et utiliser deux fonctions, une pour quand l'utilisateur demande le contenu de la variable, l'autre pour quand l'utilisateur définit le contenu de la variable. Il suffit de remplacer les deux termes REALstandardGetter/Setter par
`(REALproc)nom_de_la_fonction_en_C_Get/Set`. Les deux fonctions doivent avoir au minimum en paramètre l'instance de votre classe, comme vu à la section précédente. La fonction qui permet de gérer la définition de la valeur d'une variable doit avoir un paramètre en plus : la valeur donné par l'utilisateur, alors que la fonction qui permet de gérer le renvoi de la valeur contenue par la variable doit juste renvoyer une valeur. Il est bien à noter que vous pouvez faire ce que vous voulez du paramètre qui contient la valeur à définir ou de la valeur de retour de vos fonctions, rien n'oblige à les lier avec leur véritable variable contenue dans la structure de données codée en C/C++, mais pour avoir un Plug-In cohérent, il est mieux de bien respecter un fonctionnement logique.
Je vous explique, pour terminer la section, comment accéder aux variables contenues dans votre structure de données dans les fonctions en C/C++. C'est là que le paramètre qui contient l'instance de votre classe RealBasic

est primordial. C'est lui qui nous permet d'accéder à votre structure créée en mémoire avec l'instance de votre classe. On utilise la macro `ClassData (maClass, mon_instance, classData, me);`. Le premier argument est la variable de définition de votre classe RealBasic, le deuxième est l'instance passée en paramètre, le troisième est le nom de la structure qui contient vos variables et le dernier est le nom d'accès à votre structure. C'est un pointeur, donc pour accédez à vos variables, faites par exemple `me->ma_variable`.

### 3.2.3 Implémentation d'événements

Les événements de votre classe (qui sont disponibles quand l'instance de la classe est sous forme de contrôle), comme pour les méthodes, doivent être définis dans un tableau de définition d'événements. Par exemple :

```
REALevent classEvents[] = {
   { "EventNb2(val as integer) as integer" }, //0
   { "EventNb1(var1 as integer, var2 as boolean)" }, //1
};
```

Vous remarquez qu'il n'y a aucun lien vers du code en C/C++ dans ces définitions, ce qui est évident, car ce sont des événements, donc RB n'a pas à faire de lien vers votre code C/C++ : c'est vous qui devez faire le lien de votre code C/C++ vers les événements. Pour faire ce lien et exécuter un événement, le code n'est pas très élégant, mais une fois que vous l'avez, utilisez le copier coller, modifier deux ou trois trucs, et ça marche sans problème. Voici par exemple le code qui va exécuter le premier événement de votre classe :

```
int ret=0;
int (*fp)(REALobject, int);
fp = (int (*)(REALobject, int)) REALGetEventInstance(inst,
&classEvents[0]);
if(fp)
   ret=fp(inst, 25);
```

`ret` contient la valeur retournée dans l'event sous RealBasic, `fp` est le pointeur vers l'événement sous RealBasic et `25` est la valeur qu'on passe en paramètre de l'event (ça pourrait être tout autre valeur ou une variable). On

fait un test pour savoir si le pointeur de l'événement pointe vers rien, ce qui est le cas quand dans RealBasic il n'y a pas un seul caractère dans le code de l'événement (c'est à dire quand il n'est pas en gras dans la hiérarchie). `inst` est la variable de l'instance de votre classe (de type `REALobject`) et `&classEvents[0]` pointe sur le premier élément de votre tableau de définition d'event.

Et voici le code qui permet d'exécuter le deuxième event :

```
void (*fp)(REALobject, int, bool);
fp = (void (*)(REALobject, int, bool)) REALGetEventInstance(inst,
&classEvents[1]);
if(fp)
   fp(inst, 25, true);
```

Par ces 2 exemples, vous aurez compris le transtypage qu'il faut effectuer : l'instance, puis les paramètres qu'utilise l'event.

### 3.2.4 Définition d'une classe

Comme dit au début, il faut maintenant écrire la définition de la classe, ce qui va vous permettre de dire quelles sont les méthodes, variables, événements. . .qu'elle contient. Par exemple, pour votre classe, la définition ressemblerait à ça :

```
static REALclassDefinition maClass = {
   kCurrentREALControlVersion,
   "myClass", //Nom de la classe sous RB
   nil,       //super class
   sizeof(classData), //Taille de la structure de donnée
   0, //réservé
   nil, //constructeur
   nil, //desctructeur
   classProperties, //tableau de définition des propriétés
   sizeof(classProperties) / sizeof(REALproperty), //nombre d'éléments
   classMethods, //tableau de définition des méthodes
   sizeof(classMethods) / sizeof(REALmethodDefinition), //nbr d'éléments
   classEvents, //tableau de définition des événements
   sizeof(classEvents) / sizeof(REALevent), //nombre d'éléments
};
```

A savoir : vous pouvez étendre une classe déjà existante en mettant le même nom que la classe à étendre pour le nom de votre classe. En cas d'extension, vous ne pouvez avoir ni de constructeur ni de destructeur. Finalement, en cas d'extension, les `REALobject` sont en fait du type de la classe que vous étendez. Par exemple si vous étendez la classe String, vos `REALobject instance` dans les paramètres de vos fonctions C/C++ seront en fait des `REALstring`. Et donc vous pourrez utiliser les fonctions du SDK sur les `REALstring`, comme par exemple `REALCString(instance)`.

Une fois tout ceci fait, n'oubliez pas d'enregistrer votre classe avec la commande `REALRegisterClass(&maClass)` dans le PluginEntry.

## 3.3 Création d'un contrôle

### 3.3.1 Les contrôles sont des classes

Effectivement, un contrôle est une classe. Une classe un peu spéciale car un contrôle à l'avantage de pouvoir dessiner une représentation de ce qu'il est, et de gérer des événements utilisateur (souris, clavier, ...). Mais c'est fondamentalement les seules différences entre les deux. Et donc pour la programmation d'un contrôle, il y a très peu de différences. Ce qui change :

1. L'instance passée aux fonctions en C/C++ n'est plus de type `REALobject` mais `REALcontrolInstance`
2. L'accès à la structure de donnée se fait par `ControlData` et non plus part `ClassData`.
3. La définition

Tout le reste (tableau de méthodes, tableau de variables et structure de données, tableau d'événements, ...) ne change pas.

### 3.3.2 Définition d'un contrôle

La définition d'un contrôle ressemble de près à celle d'une classe. Par exemple :

```
REALcontrol monControl = {
   kCurrentREALControlVersion,
   "MonControle",
   sizeof(ctrlData), //Taille structure de donné
   REALinvisibleControl, //Type du contrôle
   128, // Image dans la palette
   129, // Image cliqué dans la palette. Plus d'effet sous RB 5
   32, 32, //Largeur, Hauteur du contrôle
   ctrlProperties,
   sizeof(ctrlProperties) / sizeof(REALproperty),
   ctrlMethods,
   sizeof(ctrlMethods) / sizeof(REALmethodDefinition),
   ctrlEvents,
   sizeof(ctrlEvents) / sizeof(REALevent),
   &ctrlBehaviour
};
```

Le deuxième argument est le nom du contrôle et le troisième la taille de la structure de données. Le type du contrôle permet de dire si par exemple le contrôle est invisible comme dans notre cas (les contrôles invisibles sont par exemple les Timer, Socket, etc.). On pourrait aussi par exemple régler le type en Opaque (`REALopaqueControl`). Se référer à la documentation du SDK. À savoir : vous pouvez combiner plusieurs types avec la lettre `|`. Les deux arguments suivants permettent de dire dans quelle ressource se trouvent les ima-

gettes qui vont se trouver dans la palette de contrôles (se sont des ID vers des ressources de type PICT). Ensuite vient la taille largeur par hauteur. Ça n'a pas d'influence si c'est un contrôle invisible. Puis vient ce que vous connaissez, car c'est exactement la même chose que pour les classes. J'ai juste remplacé les "classXXX" par "ctrlXXX". Le dernier paramètre `ctrlBehaviour` est par contre nouveau, et il est très important. Il pointe vers une structure qui contient uniquement des liens vers des fonctions. Cette structure permet de lier toutes sortes d'événements (événements propre à RB pour la gestion du contrôle ou événements utilisateur) à des fonctions en C/C++. Prenons la structure suivante :

```
REALcontrolBehaviour ctrlBehaviour = {
   nil,          // constructor
   nil,          // destructor
   ctrlDraw,     // redraw
   nil,          // click
   nil,          // mouse drag
   nil,          // mouse up
   nil,          // gained focus
   nil,          // lost focus
   nil,          // key down
   nil,          // control open
   nil,          // control close
   nil,          // idle
   nil,          // draw offscreen
   nil,          // set special background
   nil,          // constant changing
   nil,          // new instance dropped in IDE
   nil,          // mouse enter
   nil,          // mouse exit
   nil           // mouse move
};
```

Vous voyez alors que nous somme capable de gérer un grand nombre d'événements envoyés par RealBasic sur notre contrôle. Et je n'ai encore pas tout donné. Référez vous à la documentation du SDK pour connaître les autres possibilités d'événement. Sur mon exemple, un seul événement m'intéresse (j'aurais

d'ailleurs pu effacer ensuite tous les nils qui suivent l'événement en question, mais je les ai laissé pour vous montrer les possibilités qu'offre le SDK) : `redraw`. Il est appelé quand le contrôle est par exemple mis dans la fenêtre, qu'il est déplacé, où quand l'application est lancée et que ce n'est pas un contrôle invisible. Dans notre cas, c'est un contrôle invisible, on va donc, pour l'exemple, dessiner une image de 32 par 32 qui se trouve dans les ressources et qui représente le contrôle (comme une horloge pour le Timer par exemple). On lui a donné pour ID 500. Le code ressemblerait à ça :

```
short gMyResFile ;

static void ctrlDraw(REALcontrolInstance instance)
{
   ControlData(ctrlControl, instance, ctrlData, me) ;

   short oldResFile = CurResFile() ;
   UseResFile(gMyResFile) ;

   Rect rect ;
   REALGetControlBounds(instance, &rect) ;

   PicHandle pict ;
   pict=GetPicture(500) ;
   DrawPicture(pict, &rect) ;

   UseResFile(oldResFile) ;
}
```

puis en mettant dans la première ligne du `PluginEntry` ceci : `gMyResFile = CurResFile();` Reprenons le code, même si ce n'est pas tout à fait le but de ce document. `short gMyResFile ;` est une variable globale qui représente la zone de ressources de notre Plug-In. `oldResFile` représente la zone de ressources actuelle (c'est à dire de l'application qui utilise le PlugIn, ou de RealBasic pour les versions antérieures à la 5). `UseResFile(gMyResFile)` permet de dire aux fonctions qui utilisent les ressources d'utiliser notre zone de ressources du PlugIn, zone que nous avons stockée dans `gMyResFile` pendant que notre PlugIn était chargé en mémoire, donc avant que la zone de

ressources soit mise sur l'application qui utilise notre PlugIn. On récupère ensuite la taille et la position de notre contrôle. On récupère l'image dans notre zone de ressources et on l'affiche et enfin on re-switch sur la zone de ressources de l'application.

Une fois tout ceci fait, n'oubliez pas d'enregistrer votre contrôle avec la commande `REALRegisterControl(&monControl)` dans le PluginEntry.