Utiliser Lua en C#

Introduction

Lua est une bibliothèque open source très légère, portable, qui permet de nombreuses possibilités de script. Lua est utilisée pour des jeux commerciaux tels que *Homeworld 2*, *Blitzkrieg*, *SpellForce* et bien d'autres encore. La syntaxe du langage de script Lua ressemble fort à du Pascal, et est très simple à utiliser.

L'intérêt d'utiliser ce genre de bibliothèques est qu'elles permettent de modifier le comportement d'un programme sans avoir à le recompiler. On peut voir ça comme des fichiers de configurations externes.

Ce tutorial a pour but de vous montrer comment utiliser Lua dans vos programmes C#, qu'il s'agisse de jeux ou non. En effet, comme indiqué plus haut, Lua peut aussi servir de langage de configuration de programme.

Si vous souhaitez vous renseigner plus sur Lua, consultez le site web officiel : http://www.lua.org

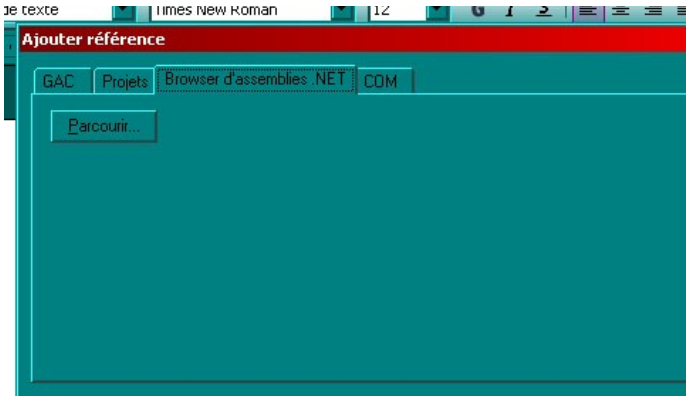
1 – Préparation

Ca y est ! Vous avez décidé d'utiliser Lua dans votre programme. Il vous faut maintenant vous procurer l'interpréteur Lua pour C#. Vous en trouverez un à cette adresse : http://luaforge.net/projects/luainterface/ . La version utilisée dans ce tutorial est la version 1.3.0.

Une fois l'archive téléchargée, décompressez-la dans le répertoire de votre choix. Dans le répertoire bin, vous trouverez les trois fichiers qui nous intéressent : `luanet.dll`, `lua50.dll`, `LuaInterface.dll`. Pour l'instant, conservez leur emplacement en tête.

Nous allons maintenant ouvrir un nouveau projet C# avec SharpDevelop. Pour faire au plus simple, nous allons faire une Application Console. Nommons-la LuaTutorial. Nous disposons donc de notre classe principale `MainClass` dans le fichier `Main.cs`.

Afin de pouvoir utiliser Lua dans notre programme, nous allons ajouter une référence vers `LuaInterface.dll`. Ajoutez une référence en cliquant droit sur Références dans l'explorateur du projet. Ensuite, sélectionnez l'onglet Browser d'assemblies .NET et cliquez sur le bouton `Parcourir...`



Sélectionnez le fichier `LuaInterface.dll` vu précédemment, et validez. La bibliothèque LuaInterface est maintenant référencée par votre projet. Il reste néanmoins à ajouter les deux autres fichiers (`luanet.dll` et `lua50.dll`) au répertoire d'exécution du programme. Recopiez ces deux fichiers dans les répertoires `bin/Debug` et `bin/Release` de votre projet.

La dernière étape consiste à indiquer dans votre fichier de code que vous allez utiliser LuaInterface.

```
using System;

using LuaInterface;

namespace LuaTutorial
{
   class MainClass
   {
      public static void Main(string[] args)
      {
         Console.WriteLine("Hello World!");
      }
   }
}
```

Nous sommes maintenant prêts à utiliser Lua dans notre programme !

2 – Premiers pas en Lua

Pour commencer, nous allons créer un interpréteur Lua. C'est assez simple...

```
Lua luaInterpret = new Lua();
```

C'est cet objet qui nous permettra d'utiliser Lua. Pour commencer doucement, nous allons créer deux variables Lua, que nous afficherons en C#. Notez bien que ces variables sont crées dans l'interpréteur Lua, et non dans le programme C#. Vous remarquerez qu'aucun type n'a été déclaré : Lua est un langage typé dynamiquement, au contraire du C#.

```
luaInterpret["chaine"] = "Une chaine";
luaInterpret["entier"] = 10;
```

Maintenant que ces variables sont créées au niveau de l'interpréteur, nous allons les récupérer pour les afficher. Lua ne travaillant qu'avec des doubles, l'entier sera récupéré sous la forme d'un double.

```
string chaine = (string) luaInterpret["chaine"];
double entier = (double) luaInterpret["entier"];
```

Il ne reste plus maintenant qu'à modifier la sortie textuelle sur la Console en utilisant nos nouvelles variables.

```
Console.WriteLine("chaine = {0}\nentier = {1}", chaine, entier);
```

Si tout va bien, vous devriez avoir en sortie le contenu des variables.

Et si nous faisions un peu travailler l'interpréteur ? Avant la sortie en Console, nous allons envoyer un ordre à l'interpréteur. Pour cela, nous allons utiliser la méthode `DoString(string chunk)`. Cette méthode prend en paramètres une chaîne de caractères correspondant à du code Lua. Généralement, le code Lua sera plutôt enregistré dans des fichiers externes à l'exécutable. Nous verrons cela plus tard, pour l'instant, donnons un bout de code brut à l'interpréteur :

```
luaInterpret.DoString("entier = entier*2;");
```

Insérez cette ligne de commande après la déclaration des variables Lua, et avant celles en C#.

Ca marche ! Nous avons maintenant 20 en sortie ! Nous allons maintenant tenter d'obtenir le même résultat à l'aide d'un fichier externe. Dans le répertoire d'exécution de votre programme (où vous avez recopié les dll), créez un répertoire scripts, et ouvrez-y un fichier texte vide que vous appèlerez `var.lua`. Voici son contenu :

```
entier = 2;
chaine = "Ma Chaine";
entier = entier*2;
```

Nous allons utiliser cette fois-ci la méthode `DoFile(string fileName)` pour charger le code Lua. Le code de la fonction `Main()` devient le suivant :

```
public static void Main(string[] args)
{
    Lua luaInterpret = new Lua();

    luaInterpret.DoFile(@"scripts/var.lua");

    string chaine = (string) luaInterpret["chaine"];
    double entier = (double) luaInterpret["entier"];

    Console.WriteLine("chaine = {0}= {1}", chaine, entier);
}
```

Voilà pour commencer ! Dans la prochaine partie, nous allons utiliser un peu de Réflexion pour utiliser nos fonctions C# dans nos scripts Lua.

3 – Fonctions et Réflexion

Avec ce que nous avons vu précédemment, il est donc possible de préparer quelques scripts basiques en Lua, qui agrèmenteront vos programmes. Néanmoins, on arrive vite à la limite de ce qui est possible.

La fonction `RegisterFunction(...)` va nous permettre d'étendre la possibilité de nos scripts. Cette fonction, en utilisant la réflexion de .NET, va enregistrer des fonctions écrites dans le programme en C# dans l'interpréteur Lua. Ceci aura pour conséquences que ces fonctions pourront être appelées dans nos scripts !

Pour commencer, nous allons créer une petite classe toute simple dans notre fichier de code :

```
class Prix
   {
      public float prixEuros;

      public Prix(float p)
      {
         prixEuros = p;
      }

      public float PrixEnFrancs()
      {
         return prixEuros*6.6557f;
      }
   }
```

Nous allons maintenant configurer l'interpréteur pour qu'il reconnaisse la fonction `PrixEnFrancs()`. La fonction `RegisterFunction()` nécessite trois paramètres. Le premier est le plus simple : il s'agit de l'alias qu'utilisera la fonction dans vos scripts. Vous pouvez reprendre le nom de la fonction, ou la renommer comme bon vous semble !

Le second paramètre est l'objet auquel est rattaché la fonction. Dans notre exemple, il s'agira d'un objet Prix nommé prix. Enfin, le troisième paramètre est une référence vers la fonction. C'est ici qu'est utilisée la réflexion. Ce tutorial n'ayant pas ce but, je ne rentrerai pas dans les détails. Sachez simplement utiliser cette méthode...

Voici le code de notre fonction `Main()` après ces quelques modifications :

```
      public static void Main(string[] args)
      {
         Prix prix = new Prix(2.15f);
         Lua luaInterpret = new Lua();
         luaInterpret.RegisterFunction("Conversion", prix,
                                       prix.GetType().GetMethod("PrixEnFrancs"));

         luaInterpret.DoFile(@"scripts/convert.lua");

         double monPrix = (double) luaInterpret["prix"];

         Console.WriteLine("Le Prix en Francs : {0} F", monPrix);
      }
```

Le fichier de script `convert.lua` ne contient qu'une seule ligne de code, qui appelle la fonction `Conversion()`, et la stocke dans la variable prix.

```
prix = Conversion();
```

Les fonctions enregistrées étant associées à une instance d'objet, il est possible par ce biais de modifier un objet. Rajoutez la fonction suivante à la classe Prix :

```
public void ModifierTarif(float prix)
      {
         prixEuros = prix;
      }
```

Cette fonction accepte un paramètre : et oui, on peut aussi enregistrer des fonctions avec des paramètres ! Attention cependant aux types utilisés. La documentation de Lua est à ce titre fort bien conçue. Vous pouvez la consultez à cette adresse : http://www.lua.org/pil

Il nous reste donc à enregistrer cette fonction de la même façon que la précédente. Seul le nom de la fonction est nécessaire pour l'enregistrer, même si elle possède des paramètres. Attention cependant si vous utilisez le polymorphisme : une exception du type `AmbiguousMatchException` sera levée.

```
luaInterpret.RegisterFunction("ModifierTarif", prix,
                              prix.GetType().GetMethod("ModifierTarif"));
```

N'oubliez pas d'appeler cette nouvelle fonction dans votre script :

```
ModifierTarif(1.50);
prix = Conversion();
```

4 – Lua orienté-objet

Maintenant que nous avons vu comment enregistrer nos fonctions C# pour les utiliser dans les scripts Lua, il est temps de passer à la vitesse supérieure. En effet, la limite imposée par la méthode précédente est la présence d'un objet cible auquel on rattache les fonctions. Il n'est pas possible de manipuler directement les objets. Nous allons nous intéresser à ça dans cette partie.

Pour commencer, modifions la classe Prix pour lui rajouter une dénomination, ainsi qu'une fonction de ristourne :

```
public string denomination = "";

public void Ristourne()
{
   prixEuros *= 0.75f;
}
```

Nous allons ensuite créer une classe Facture qui accueillera plusieurs objets Prix (dans un ArrayList), via une méthode que nous enregistrerons. N'oubliez pas de rajouter la référence à `System.Collections` :

```
using System.Collections;
```

```
   class Panier
   {
      public ArrayList liste;

      public Panier()
      {
         liste = new ArrayList();
      }

      public void AjouterPrix(Prix p)
      {
         liste.Add(p);
      }
   }
```

Pour pouvoir utiliser des objets C# dans nos scripts Lua, il faut commencer par charger les assemblages où se trouvent ces objets. On utilise pour cela la fonction `load_assembly`. Attention ! Cette fonction n'est pas une fonction C#, mais une fonction Lua. Elle s'utilise de la manière suivante :

```
luanet.load_assembly("LuaTutorial");
```

La chaîne entre parenthèses correspond au namespace dans lequel se trouvent nos classes. Vous pouvez utiliser les classes de base de C# en indiquant également le namespace dans lequel elles se trouvent.

Maintenant que l'assemblage a été indiqué, il faut enregistrer le type. Là aussi, il s'agit d'une fonction Lua, nommée `import_type`. Voici son utilisation :

```
Prix = luanet.import_type("LuaTutorial.Prix");
```

Comme lors de la définition des fonctions, le nom que vous donnez au type créé n'a que peu d'importance. Il peut être différent du nom de la classe originale.

Désormais, vous pouvez utiliser ce type dans vos scripts Lua en utilisant les mêmes fonctions qu'en C#. Voici ce que donne le script Lua que nous nommons panier.lua :

```
luanet.load_assembly("LuaTutorial");

Article = luanet.import_type("LuaTutorial.Prix");
p1 = Article(4.0);
p1.denomination = "Premier article";
p1:Ristourne();
AjouterArticle(p1);

p2 = Article(145.5);
p2.denomination = "Deuxieme article";
AjouterArticle(p2);
```

La création de l'objet se fait presque comme en C#, à ceci près qu'on n'utilise pas le mot-clé `new`. Pour l'accès aux variables, il faut utiliser un point suivi du nom de la variable. Pour les fonctions, il s'agit de deux points `":"`. Les fonctions utilisées ainsi n'ont pas à être enregistrées au préalable. L'interpréteur s'occupe de leur faire correspondre la bonne fonction.

Avec toutes ces modifications, notre fonction principale n'est plus très à jour. Il nous faut enregistrer la fonction `AjouterArticle`, créer un panier, et exécuter le nouveau script. Voici sa nouvelle apparence :

```
Panier panier = new Panier();
Lua luaInterpret = new Lua();
luaInterpret.RegisterFunction("AjouterArticle", panier,
                              panier.GetType().GetMethod("AjouterArticle"));

luaInterpret.DoFile(@"scripts/panier.lua");

foreach(Prix p in panier.liste)
   {
      Console.WriteLine("{0}euros : {1}francs : {2}",
                        p.denomination,p.prixEuros,p.PrixEnFrancs());
   }
```

Et voilà, on arrive au bout du chemin ! Pour le dernier bout de code à taper, je vous laisse maître. Sans modifier le programme C# (sans le recompiler, donc), modifiez le résultat obtenu, en ajoutant de nouveaux articles, en effectuant des ristournes, et autres...

Conclusion

C'est la fin de ce tutorial, qui, je l'espère, vous a permis de maîtriser les bases de l'utilisation de Lua en C#. Maintenant que vous voyez les possibilités offertes, lâchez-vous !

Crédits

Ce tutorial a été rédigé par Gulix (gulix33xp@yahoo.fr)

Il est disponible dans sa dernière version sur http://gulix.free.fr

Une version est également disponible sur le wiki de http://www.games-creators.org

Un remerciement tout particulier à Dan (http://www.godpatterns.com), dont les tutoriaux m'ont guidé pour rédiger celui-ci.

Dernière mise-à-jour : le 13 février 2006