Utilisation de Lua comme langage de script - Partie 2 : Utiliser Lua avec des classes C++

Par Michel de VERDELHAN

Date de publication : 24 novembre 2006

Nous avons vu dans le tutoriel précédent comment faire pour appeler des fonctions Lua depuis le C et vice versa. Dans ce tutoriel, nous allons nous intéresser à l'utilisation de classes C++, et à leur binding en Lua avec Lunar.

I - Introduction

A la base, Lua ne permet pas d'utiliser des objets dans ses scripts. Malheureusement, pour développer un jeu, on a quasiment toujours besoin de classes ou au moins de structures qui vont nous permettre de manipuler nos différents objets. Heureusement pour nous, Lua offre un mécanisme de méta tables qui permet d'étendre les fonctionnalités du langage. Le problème est que ce système de méta tables n'est pas facile à utiliser... c'est pour cela que nous allons utiliser un binding (1) simple entre le C++ et Lua qui va prendre en charge les modifications des méta tables à notre place.

Il existe de nombreux binding de Lua pour le C++. Cela va du binding très simple comme Luna (disponible sur le Wiki officiel de Lua) qui ne permet pas de faire grand-chose, au binding très complexe comme LuaBind qui, lui, permet une utilisation poussée de Lua avec C++ (transfert de gestion de la mémoire, définition de classes directement en Lua...)

Dans ce tutoriel, nous utiliserons une version améliorée de Luna qui s'appel Lunar. Contrairement à Luna, cette version permet de passer des classes C++ comme paramètres à Lua, ce qui va bien entendu nous servir dans de nombreuses classes C++/Lua.

II - Présentation de Lunar

Le binding que nous allons utiliser dans ce tutoriel est un wrapper C++ appelé Lunar qui est disponible sur le Wiki officiel de Lua à l'adresse **http://lua-users.org/wiki/CppBindingWithLunar**. Ce binding est très simple à utiliser, il suffit d'ajouter un fichier .h dans le répertoire contenant déjà les fichiers de Lua, et voilà, on peut utiliser Lunar.

Lunar est un en fait une classe template très simple qui offre juste quatre services à l'utilisateur :

1. Enregistrer une classe auprès de Lua grâce à la méthode **Register**. Cette méthode doit être appelée après la création du contexte Lua. Elle prend en paramètre le contexte dans lequel la classe sera disponible. On peut donc avoir plusieurs scripts lancés en même temps dont certains ne peuvent pas accéder à la classe alors que les autres le peuvent. Ceci peut être utile pour, par exemple, différencier les scripts qui touchent aux moteurs (3D, IA...) des scripts traitant du déroulement du jeu.
2. Appeler une méthode donnée grâce à la méthode **call**. Cette méthode prend en paramètre le contexte d'exécution et le nom de la méthode à appeler. Elle a aussi trois paramètres facultatifs :
   - Le nombre d'arguments de la méthode (valeur par défaut : 0).
   - Le nombre de valeurs de retour. Par défaut, la méthode peut retourner plusieurs valeurs.
   - Le numéro de la fonction de gestion des erreurs (valeur par défaut : 0).
3. Empiler un objet avec sa méta table grâce à la méthode **push**. Cette méthode prend en paramètres le contexte d'exécution, un pointeur vers l'objet à empiler et un booléen facultatif permettant de définir si on souhaite que le ramasse-miettes de Lua gère l'objet ou pas (par défaut, l'objet n'est pas géré par le ramasse-miettes).
4. Récupérer un objet sur la pile grâce à la méthode **check**. Cette méthode prend en paramètres le contexte d'exécution de Lua et l'indice de l'objet à récupérer dans la pile. Elle permet aussi de vérifier que l'objet récupéré est du bon type. Elle retourne un pointeur vers l'objet récupéré sur la pile. Si l'objet n'est pas du bon type, elle retourne NULL.

Pour illustrer les différentes possibilités offertes par Lunar, nous allons utiliser dans ce tutoriel une classe de vecteur 3D simple dont voici le code qui sera détaillé par la suite:

Classe de vecteur 3D utilisée dans le tutoriel.

```
class Vector3
{
private:
 friend class Lunar<Vector3>;
 static const char className[];
 static Lunar<Vector3>::RegType methods[];
 float x,y,z;
public:
 Vector3()
 {
  x=y=z=0;
 }
 Vector3(float x, float y, float z)
 {
  this->x = x;
  this->y = y;
  this->z = z;
 }
 Vector3(lua_State* L)
 {
  int nbArgs = lua_gettop(L);
  if (nbArgs == 3 && lua_isnumber(L,1) &&
   lua_isnumber(L,2) && lua_isnumber(L,3))
  {
   this->x = lua_tonumber(L,1);
   this->y = lua_tonumber(L,2);
   this->z = lua_tonumber(L,3);
  }
  else if (nbArgs == 0)
```

Classe de vecteur 3D utilisée dans le tutoriel.

```
  {
   x = y = z = 0;
  }
  else
  {
   cerr << "Vector3 : mauvais paramètres" << endl;
  }
 }

 void set(float x, float y, float z)
 {
  this->x = x;
  this->y = y;
  this->z = z;
 }

 int set(lua_State* L)
 {
  int nbArgs = lua_gettop(L);
  if (nbArgs == 3 && lua_isnumber(L,1) &&
   lua_isnumber(L,2) && lua_isnumber(L,3))
  {
   this->x = lua_tonumber(L,1);
   this->y = lua_tonumber(L,2);
   this->z = lua_tonumber(L,3);
  }
  else if (nbArgs == 0)
  {
   x = y = z = 0;
  }
  else
  {
   cerr << "Vector3:set() : mauvais paramètres" << endl;
  }
  return 0;
 }
 int print(lua_State* L)
 {
  cout << "{" << x << "," << y << "," << z << "}" << endl;
  return 0;
 }
 float length()
 {
  return sqrt(x*x + y*y + z*z);
 }
 int length(lua_State* L)
 {
  lua_pushnumber(L,length());
  return 1;
 }
 int add(lua_State* L)
 {
  Vector3* param = Lunar<Vector3>::check(L,1);
  if (param == NULL)
  {
   cerr << "Vector3:add() : mauvais type de paramètre" << endl;
   return 0;
  }
  Vector3* ret = new Vector3(x+param->x,y+param->y,z+param->z);
  // on souhaite que l'objet soit géré par le ramasse-miettes
  Lunar<Vector3>::push(L,ret,true);
  return 1;
 }
 int equal(lua_State* L)
 {
  Vector3* param = Lunar<Vector3>::check(L,1);
  if (param == NULL)
  {
   cerr << "Vector3:equal() : mauvais type de paramètre" << endl;
   return 0;
```

Classe de vecteur 3D utilisée dans le tutoriel.

```
   }
   bool ret = x==param->x && y==param->y && z==param->z;
   lua_pushboolean(L,ret);
   return 1;
  }
  string toString()
  {
   std::stringstream s;
   s << "{" << x << "," << y << "," << z << "}";
   return s.str();
  }
};

// fonction pour pouvoir afficher le vecteur simplement sur un flux de données.
std::ostream& operator<<(std::ostream& stream, Vector3& vector)
{
 return stream << vector.toString();
}
```

Comme vous pouvez le voir, cette classe contient des méthodes qui ressemblent aux fonctions Lua qu'on a vu au tutoriel précédent. Ce sont nos méthodes accessibles en Lua. Nous allons voir maintenant comment faire pour pouvoir les utiliser effectivement dans un script Lua.
Il y a certains points dans cette classe qui n'ont pas encore été abordés dans ce tutoriel ni dans le précédent. Ne vous inquiétez pas, ils seront traités dans la suite de ce tutoriel.

III - Enregistrement et définitions de méthodes en Lua

Dans cette partie, nous allons voir comment faire pour pouvoir enregistrer une classe auprès de Lua et comment faire pour enregistrer les méthodes de cette classe en Lua. Nous verrons aussi comment créer des méthodes directement en Lua.

III-1 - Enregistrer une classe auprès de Lua

Pour pouvoir enregistrer une classe avec Lunar, il faut que cette classe définisse deux choses :

- Une chaîne de caractères contenant le nom utilisé en Lua pour notre classe (on peut donner un nom différent de celui utilisé en C++). Cette chaîne doit obligatoirement s'appeler *className* pour que Lunar puisse la retrouver.
- Un tableau de descripteurs de méthodes qui sera utilisé pour charger les méthodes de la classe dans la méta table Lua. Ce tableau doit obligatoirement s'appeler *methods* pour que Lunar puisse le retrouver.

Voila l'entête nécessaire pour pouvoir utiliser notre classe de vecteur avec Lunar :

Entête obligatoire pour pouvoir utiliser notre classe avec Lunar.

```
class Vector3
{
private:
 friend class Lunar<Vector3>;
 static const char className[];
 static Lunar<Vector3>::RegType methods[];
};
```

Il nous faut maintenant définir le nom de notre classe et l'ensemble des méthodes qui seront accessibles depuis Lua. Pour cela, il suffit de faire le bout de code suivant :

Définition du nom de la classe et des méthodes utilisables en Lua.

```
const char Vector3::className[] = "Vector3";

#define method(class, name) {#name, &class::name}

// on initialise le tableau des méthodes de la classe.
Lunar<Vector3>::RegType Vector3::methods[] = {
  method(Vector3, methodeName1),
  method(Vector3, methodeName2),
  method(Vector3, methodeName3),
  {0,0}
};
```

Il ne nous reste plus qu'à enregistrer notre classe auprès de Lua comme ceci :

Enregistrement de la classe auprès de Lua.

```
// on enregistre la classe auprès de lua
Lunar<Vector3>::Register(state);
```

Maintenant, nous pouvons appeler des méthodes sur des objets en Lua comme ceci :

```
u = Vector3(10,20,30);
u:print();
```

Notez que Lunar ne permet d'accéder qu'à des méthodes, les attributs ne peuvent donc êtres manipulés que depuis des accesseurs/modificateurs.

III-2 - Enregistrer une méthode auprès de Lua

Je vais revenir un peu sur comment faire pour pouvoir enregistrer une méthode pour qu'elle soit utilisable en Lua. Comme pour les fonctions du premier tutoriel, il faut que la méthode utilise la signature suivante :

Signature d'une méthode appelable depuis Lua.

```
int methodeName(lua_State* L);
```

Ensuite, il ne faut pas oublier de l'ajouter au tableau des méthodes de la classe pour que Lunar puisse la trouver au moment de l'enregistrement auprès de Lua. Pour cela, il suffit de rajouter une ligne lors de l'initialisation du tableau de méthodes. Par exemple, pour ajouter la méthode *cross*, qui effectue le calcul du produit vectoriel, à notre classe de vecteur 3D, il suffit d'ajouter la ligne suivante :

Ajout de la méthode cross à notre classe.

```
#define method(class, name) {#name, &class::name}

Lunar<Vector3>::RegType Vector3::methods[] = {
  method(Vector3, set),
  method(Vector3, print),
  method(Vector3, length),
  method(Vector3, cross), // <= la méthode qu'on ajoute.
  {0,0}
};
```

Si vous utilisez la macro défini juste avant le tableau comme nous venons de le faire, il est important de bien utiliser le nom de la méthode pour l'ajouter au tableau. Sinon, vous pouvez faire comme ceci :

Autre technique pour ajouter une méthode.

```
#define method(class, name) {#name, &class::name}

Lunar<Vector3>::RegType Vector3::methods[] = {
  method(Vector3, set),
  method(Vector3, print),
  method(Vector3, length),
  {"cross",&Vector3::cross}, // <= la méthode qu'on ajoute.
  {0,0}
};
```

Ceci permet de définir un nom de méthode Lua différent de celui utilisé en C++.

III-3 - Définir/redéfinir une méthode en Lua

Grâce à Lunar, on peut ajouter des méthodes à nos classes directement en Lua, mais on peut aussi surcharger une méthode C++ en Lua.

Ceci peut être pratique quand on souhaite que ce soit le programmeur du script qui définisse le comportement de la méthode, tout en étant sûr que la méthode existe bien. On peut utiliser ce mécanisme pour définir des méthodes callback. Par exemple, si on a une classe Ennemy qui gère nos ennemis, qu'on souhaite que notre ennemi réagisse quand il voit le joueur, mais qu'on ne souhaite pas définir sa réaction en C++, on peut créer une méthode callback (appelons la onSeePlayer), et l'utilisateur final va pouvoir redéfinir cette méthode directement en Lua. Ensuite, lorsqu'un ennemi aperçoit le joueur, il ne nous reste plus qu'à appeler la méthode Lua. Si la méthode a bien été redéfinie par l'utilisateur, c'est cette méthode qui sera appelée, sinon, c'est la méthode C++ qui sera appelée. Notez

bien qu'il faut appeler la méthode Lua et non pas la méthode C++ ici. Si vous appelez la méthode C++, ce sera effectivement elle qui sera appelée même si elle a été redéfinie en Lua...

Maintenant que nous savons à quoi peut bien servir la redéfinition de méthode, il ne nous reste plus qu'à voir comment faire pour définir une méthode en Lua. C'est très simple, il suffit de définir une fonction dont le nom commence par le nom de la classe suivi de deux points comme ceci :

Redéfinition de méthode en Lua.

```
-- redéfinition de la méthode print du vecteur.
function Vector3:print()
 print("redéfinition de la méthode print de Vector3");
 print(self:length());
end
```

Les méthodes de l'objet peuvent êtres accédées via l'opérateur *self* de Lua qui est l'équivalent du *this* en C++.

Maintenant que nous savons comment définir des méthodes C++ utilisable en Lua, il ne nous reste plus qu'à les appeler. Ceci se fait comme en C++ mais avec l'opérateur ":" au lieu des opérateurs "." ou "->". Voici un exemple simple d'appel de la méthode add d'un vecteur :

```
v1 = Vector3(1,2,3);
v2 = Vector3(5,6,7);
v3 = v1:add(v2);
```

IV - Appels de méthodes en Lua

Dans cette partie, nous allons voir comment appeler une méthode Lua en C++.

IV-1 - Appeler une méthode sur un objet précis

Comme nous venons de le voir dans la partie III-3, il peut être pratique de redéfinir une méthode C++ en Lua, mais pour que ceci soit utile au programmeur C++, il faut appeler la méthode Lua depuis le C++. C'est ce que nous allons voir ici.

Appeler une méthode avec Lunar se fait en deux étapes :

- Empiler l'objet sur lequel sera effectué l'appel de méthode grâce à *Lunar<>::push()*.
- Appeler la méthode avec *Lunar<>::call()* qui va effectivement effectuer l'appel de la méthode.

Voici un exemple d'appel de méthode simple avec Lunar :

Appel d'une méthode définie dans le script Lua.

```
// appel d'une méthode Lua définie dans le script.
lua_settop(state,0);
Vector3 vect(12,13,14);
Lunar<Vector3>::push(state,&vect);
Lunar<Vector3>::call(state,"printLength",0,0);
```

IV-2 - Appeler une méthode avec des paramètres simples

Comme dans le tutoriel précédent, l'appel de méthode avec des paramètres se fait en empilant les paramètres sur la pile. Pour les paramètres simples (types de données Lua de base, c'est-à-dire : les nombres, les booléens et les strings), il suffit d'utiliser les fonctions Lua standard. Voici un exemple d'appel de méthode avec des paramètres simples :

Appel d'une méthode avec paramètres.

```
// appel d'une méthode avec paramètres
Vector3 vect(10,20,30);
lua_settop(state,0);
Lunar<Vector3>::push(state,&vect);
lua_pushnumber(state,1);
lua_pushnumber(state,1);
lua_pushnumber(state,1);
Lunar<Vector3>::call(state,"set",3,0);
// on affiche le résultat
cout << vect << endl;
```

IV-3 - Appeler une méthode avec des objets en paramètres

Pour appeler une méthode prenant des objets en paramètres, il suffit d'utiliser la méthode *Lunar<>::push()* qui va empiler l'objet sur la pile comme n'importe quel autre type standard de Lua. On peut donc appeler une méthode prenant en paramètres des objets comme ceci :

Appel d'une méthode avec un objet en paramètre.

```
// appel d'une méthode avec un objet comme paramètre
lua_settop(state,0);
Vector3 v1(10,10,10);
Vector3 v2(10,10,10);
// on empile le vecteur à appeler
```

Appel d'une méthode avec un objet en paramètre.

```
Lunar<Vector3>::push(state,&v1);
// on empile le paramètre
Lunar<Vector3>::push(state,&v2);
Lunar<Vector3>::call(state,"equal",1,1);
// on test le retour.
if (lua_isboolean(state,-1))
{
 bool ret = lua_toboolean(state,-1);
 if (ret)
 {
  cout << "les deux vecteur sont égaux" << endl;
 }
 else
 {
  cout << "les deux vecteur ne sont pas égaux" << endl;
 }
}
else
{
 cerr << "problème lors de l'appel à v1.equal(v2)" << endl;
}
```

IV-4 - Récupérer une valeur de retour d'une méthode

Pour récupérer une valeur de retour, on passe encore une fois par la pile. Le code ne change pas de celui utilisé pour une fonction vue dans le premier tutoriel. Ça donne donc :

Appel d'une méthode et traitement du retour.

```
// appel d'une méthode avec retour
Vector3 vect(10,20,30);
lua_settop(state,0);
Lunar<Vector3>::push(state,&vect);
Lunar<Vector3>::call(state,"length",0,1);
if (lua_isnumber(state,-1))
{
 float ret = lua_tonumber(state,-1);
 cout << "resultat de l'appel à vect.length() : " << ret << endl;
 if (ret == vect.length())
 {
  cout << "Lua à bien retourné le même resultat que le C++" << endl;
 }
 else
 {
  cerr << "Lua n'a pas retourné le même résultat que le C++" << endl;
 }
}
else
{
 cerr << "problème lors de l'appel à vect.length()" << endl;
}
```

Dans le cas où on doit récupérer un objet en retour, il faut utiliser la méthode *Lunar<>::check()* pour vérifier si l'objet retourné est bien du type voulu. Voici un exemple d'utilisation de cette méthode dans la méthode add() de notre vecteur :

Exemple d'utilisation de Lunar<>::check().

```
int add(lua_State* L)
{
 Vector3* param = Lunar<Vector3>::check(L,1);
 if (param == NULL)
 {
  cerr << "Vector3:add() : mauvais type de paramètre" << endl;
  return 0;
```

Exemple d'utilisation de Lunar<>::check().

```
  }
  Vector3* ret = new Vector3(x+param->x,y+param->y,z+param->z);
  // on souhaite que l'objet soit géré par le ramasse-miettes
  Lunar<Vector3>::push(L,ret,true);
  return 1;
 }
```

V - Conclusion

Grâce à Lunar, nous pouvons maintenant utiliser des classes dans notre code Lua. Néanmoins, ceci présente quand même le désavantage de doubler le code C++ requis pour nos classes. En effet, pour chaque méthode d'une classe qu'on souhaite voir accessible aussi bien en C++ qu'en Lua, il faut soit ne faire appel aux méthodes que via Lua, ce qui oblige à faire du code complexe pour un simple appel de méthode, soit définir deux versions de chaque méthode : une pour Lua et une pour le C++. De même, il nous est impossible avec Lunar d'avoir accès directement aux attributs de nos classes. Nous sommes obligé de passer par des accesseurs/modificateurs.

Pour résoudre ces problèmes, une solution peut être d'utiliser un autre wrapper, LuaBind, qui permet de réutiliser le code C++ pour générer le code Lua très simplement.

Vous pouvez télécharger les sources de ce tutoriel **ici** ou **ici [http]**
Une version PDF de ce tutoriel est disponible **ici** ou **ici [http]**

Tutoriel 1 : Utiliser Lua dans du code C **Retour au menu.**

1 : Un binding consiste à lier deux éléments entre eux. Dans notre cas, binder Lua avec du C++ consiste à crée un lien entre le code Lua et le code C++.