ARéVi : Atelier de Réalité Virtuelle

Guide d'utilisation

Fabrice Harrouet — (CERV, LI2/ENIB)

— Août 2006 —


Centre Européen de Réalité Virtuelle
Fabrice HARROUET
e-mail : harrouet@enib.fr
url : http://www.enib.fr/~harrouet/
tel : +33 (0)298 05 89 44
fax : +33 (0)298 05 89 79
École Nationale d'Ingénieurs de Brest
Laboratoire d'Ingénierie Informatique
Technopôle Brest-Iroise
Site de la Pointe du Diable
Parvis Blaise PASCAL
29280 Plouzané, Finistère
Adresse postale : C.S. n° 73862, 29238 Brest Cedex 3, France

Avant-propos

Cette documentation concerne l'utilisation de la bibliothèque *ARéVi* (**A**telier de **Ré**alité **Vi**rtuelle) dans l'état ou elle se trouve à la date du 02 août 2006. Cette bibliothèque, disponible selon la license *LGPL*[α] et librement téléchargeable[β], propose des facilités pour concevoir des applications à base d'entités autonomes dans des univers tridimensionnels.

Ce document est un guide d'utilisation et non un manuel de référence, c'est à dire que les principes d'utilisation des services proposés sont présentés mais que les diverses fonctions ou méthodes ne sont pas expliquées dans le moindre détail. En effet, un soin particulier a été apporté à la *"lisibilité"* des fichiers d'en-tête (`".h"`) puisque :

nous supposons nous adresser à une audience de programmeurs, à savoir des personnes qui maîtrisent *C++* et surtout qui savent lire des déclarations et interpréter les signatures.

Ainsi, le texte de ce document fait très souvent référence à des classes ou à des méthodes en évoquant simplement leur nom. Le lecteur devra se reporter au fichier d'en-tête correspondant afin d'obtenir tous les détails. Soulignons, encore une fois, que si les fonctionnalités et les principes sont correctements expliqués dans ce document, l'interprétation des fichiers d'en-tête ne doit poser aucun problème à un programmeur. De plus, le répertoire `Tests` qui accompagne *ARéVi* renferme de nombreux programmes de tests qui peuvent servir d'exemple pour débuter l'apprentissage de cette bibliothèque.

L'utilisation d'un outil de documentation *"en ligne"* consistant à insérer du texte dans les fichiers d'en-tête n'a pas été retenue afin de préserver la lisibilité de ceux-ci (chère aux programmeurs). De plus, le fait de préciser que la méthode `getX()` renvoie la valeur de `x` n'apporte pas grand chose à la compréhension des principes d'utilisation de la classe concernée et des rôles qu'elle joue vis à vis des autres classes.

[α] voir `http://www.gnu.org/copyleft/lesser.html`

[β] sur `http://www.enib.fr/~harrouet/`

Concernant son implémentation, la bibliothèque *ARéVi* ne repose que sur *C++*, *POSIX*, *OpenGL*, *X11/Win32* afin de limiter les dépendances et ainsi faciliter la portabilité. Toutefois, pour ne pas se priver de fonctionnalités avancées qui peuvent être disponibles, sous formes de bibliothèques, sur la plateforme de développement, un système de *plugins* permet de rendre ces services accessibles sans créer de dépendance forte. Ainsi, *ARéVi* est accompagnée de *plugins* reposant sur :

- *ImageMagick* (chargement d'images)
- *Imlib2* (chargement d'images)
- *FFmpeg* (géneration de vidéos)
- *ESD* (capture/jeu de son)
- *OpenAL* (son 3D)
- *Cg* (*pixel/vertex shaders*)
- *LibXml2* (lecture/écriture de flux XML)
- *ElanSpeech* (synthèse vocale)
- *JpegLib* (compression/décompression d'images)
- *ZLib* (compression/décompression de données)

Si la plateforme ne dispose pas des services/bibliothèques nécessaires pour un *plugin* particulier, ce dernier ne sera pas utilisable mais les autres services d'*ARéVi* seront tout de même disponibles. Une application *ARéVi* peut également jouer le rôle d'un *plugin* pour une application en *Tcl/Tk* ou en *Java* ; une communication bidirectionnelle est alors disponible entre *ARéVi* et le langage en question. Des *IHM* peuvent être adjointes grâce au lien avec *Tcl/Tk* ou *Java* mais également à l'aide de bibliothèques dédiées telles que *Gtk+*, *Qt* ... Pour l'instant, *ARéVi* a été testé sur les quelques plateformes dont nous disposons :

- *Linux/ix86/x86-64/PPC*
- *Window$/Cygwin* (affichage *Win32*)
- *MacOsX 10.4* (affichage *X11*)
- *IRIX 6.5*
- *FreeBSD 5.2*

L'utilisation sur d'autres plateformes ne devrait toutefois pas poser de problèmes autres que le choix des options de compilation adaptées. Certains plugins ne sont pas encore fonctionnels sur toutes les plateformes pour le simple raison que les bibliothèques requises ne sont pas forcément disponibles.

Table des Matières

Liste des Figures

Chapitre 1

Mécanismes centraux

Ce chapitre traite des mécanismes principaux de la bibliothèque *ARéVi* impliquant une démarche et un style d'écriture homogène qu'il faudra veiller à respecter pour tous les services et toutes les applications d'*ARéVi*.

Les types fournis se décomposent en trois catégories :

- Les `ArRef<T>`, `ArConstRef<T>`, `ArPtr<T>` et `ArConstPtr<T>`, qui représentent des substituts de pointeurs, permettant entre autre de bénéficier d'un ramasse-miettes.
- Une hiérarchie de classes dont l'ancètre commun est `ArObject` et dont les instances doivent être manipulées par des `ArRef<T>`, `ArConstRef<T>`, `ArPtr<T>` et `ArConstPtr<T>`.
- Le singleton `ArSystem`, qui sert à l'initialisation de l'application, à l'encapsulation des services de programmation système, et à l'exécution de la boucle de simulation.

1.1 Les pseudo-pointeurs

Un point qui motive souvent des reproches à l'encontre de *C++* concerne l'utilisation des pointeurs et en particulier la politique d'allocation/libération d'objets qui sont eux-mêmes partagés par d'autres objets. De nombreux langages proposent des mécanismes de ramasse-miettes afin de décharger l'utilisateur de cette tâche. De la même façon, en *C/C++*, il existe de nombreuses solutions à base de *smart-pointers* plus ou moins souples et automatisées.

ARéVi utilise le mécanisme *template* afin de fournir des types qui se substituent complètement aux pointeurs et qui s'utilisent exactement de la même façon. Le propos principal de ces pseudo-pointeurs est de pouvoir mettre en œuvre un ramasse-miettes qui soit à la fois automatique et contrôlable.

1.1.1 Les ArRef<T>/ArConstRef<T>

Le type `ArRef<T>` représente donc une référence sur un objet de type `T`. Ceci signifie qu'il désigne un objet que l'on peut manipuler grâce à l'opérateur `->` exactement comme on le ferait avec un pointeur. L'objet désigné est considéré comme étant *"référencé"* c'est à dire qu'il ne sera pas automatiquement détruit par le ramasse-miettes tant qu'il sera désigné par au moins un `ArRef<T>`.

Attention, le type `T` cité ici est supposé correspondre à la classe `ArObject` ou à une classe qui en est dérivée (voir le paragraphe 1.2 de la page 7).

De la même façon, le type `ArConstRef<T>` représente une référence constante sur un objet de type `T`. Ceci signifie qu'à travers cette référence, il est impossible d'invoquer des operations qui modifient l'objet désigné ; seules les opérations qualifiées comme étant `const` sont autorisées. Nous retrouvons ici la même distinction entre `ArRef<T>` et `ArConstRef<T>`, qu'entre `T *` et `const T *`.

Ces `ArRef<T>`/`ArConstRef<T>` sont constitués en interne d'un unique pointeur. Ils peuvent donc être passés par recopie lors des appels de fonctions, et se comportent de la même façon que les pointeurs en ce qui concerne :

- ▷ la recopie,
- ▷ l'affectation,
- ▷ la comparaison,
- ▷ le test de valeur nulle,
- ▷ l'accès à l'objet désigné.

Les initialisations, affectations et comparaisons doivent bien entendu avoir lieu pour des objets de types compatibles et en respectant l'indication constant/non-constant (exactement comme pour les pointeurs en *C++*). Il existe de plus des opérations spécifiques à ces références :

- ▷ l'initialisation à la valeur nulle par défaut,
- ▷ l'affectation à la valeur nulle,
- ▷ la destruction explicite de l'objet désigné,
- ▷ le test de validité de l'objet.

La figure 1.1 ci-contre illustre l'utilisation de toutes ces opérations.

Bien qu'il y ait un mécanisme de ramasse-miettes qui permette de détruire automatiquement les objets qui ne sont plus référencés par des `ArRef<T>`/`ArConstRef<T>`, il est possible de détruire explicitement un objet alors qu'il est toujours référencé. Pour ceci, il est interdit d'utiliser l'opérateur `delete` de *C++* ; il faut avoir recours à la

```
ArRef<MyClass> r1;          // initialisation implicite avec la valeur nulle
ArRef<MyClass> r2( ... );   // initialisation explicite
ArRef<MyClass> r3= ... ;    // initialisation explicite
ArRef<MyClass> r4=r2;       // initialisation par recopie
ArRef<MyClass> r5(r3);      // initialisation par recopie

r2=r3;                      // r2 désigne maintenant le même objet que r3

if(r4==r2) { ... }          // r4 et r2 désignent le même objet ?
if(r5!=r3) { ... }          // r5 et r3 ne désignent pas le même objet ?

if(r2) { ... }              // r2 n'a pas la valeur nulle, il désigne un objet ?
if(!r1) { ... }             // r1 a la valeur nulle, il ne désigne pas d'objet ?

r5->myMethod();             // appel d'une méthode sur l'objet désigné par r5
r5->myAttribute++;          // accès à un attribut de l'objet désigné par r5

r2.clear();                 // r2 ne désigne plus d'objet, il a la valeur nulle

r3.destroy();               // destruction explicite de l'objet désigné par r3

if(r4.valid()) { ... }      // r4 désigne un objet, et celui-ci n'est pas
                            // en cours de destruction ?
```

Figure 1.1 : Opérations usuelles sur les pseudo-pointeurs

méthode `destroy()` de `ArRef<T>`. Lorsqu'un objet doit être détruit, que se soit automatiquement ou explicitement, la séquence suivante a lieu :

- ▷ l'objet commence par se marquer comme étant *"invalide"* (la méthode `valid()` des `ArRef<T>`/`ArConstRef<T>` qui le désignent renvoie désormais `false`),
- ▷ il génère un événement signalant sa disparition imminente (voir le paragraphe 1.4 de la page 24),
- ▷ il se détruit effectivement (appel du destructeur *C++*),
- ▷ il finit par affecter tous les `ArRef<T>`/`ArConstRef<T>` qui le désignaient à la valeur nulle afin qu'ils ne le désignent plus.

Ceci permet donc de s'assurer du fait qu'à aucun moment il n'est possible de désigner un objet qui a été détruit (source de nombreux effets de bords indésirables en *C++*). En cas de doutes sur l'existence d'un objet que l'on référence, il suffit de tester si cette référence n'est pas devenue nulle. Dans le pire des cas, si ce test n'est pas effectué, la référence est utilisée alors qu'elle est devenue nulle, il se produit un plantage *"franc"* (utilisation d'un pointeur nul) qui ne provoque pas d'effet de bord incontrôlable et qui est extrêmement facile à repérer avec un *debugger*.

L'équivalent de `(T *)0` pour un `ArRef<T>` est obtenu par l'appel à la méthode statique en ligne `T::nullRef()`. L'équivalent de `this` pour un `ArRef<T>` est obtenu par l'appel à la méthode en ligne `thisRef()`.

Pour assurer la cohérence de l'ensemble, il est très fortement recommandé de proscrire l'utilisation des `T *`/`const T *`. On les remplacera presque systématiquement par des `ArRef<T>`/`ArConstRef<T>`.

1.1.2 Les ArPtr<T>/ArConstPtr<T>

Pour cette raison, il a été rendu *"relativement inconfortable"* de passer des pointeurs aux ArRef<T>/ArConstRef<T> et inversement. Cela a toutefois été laissé possible au cas où les détails d'implémentation d'une application particulière le nécessiteraient absolument. On obtient le T * contenu dans un ArRef<T>, et respectivement le const T * contenu dans un ArConstRef<T>, en invoquant la méthode c_ptr() de la référence. Pour créer un ArRef<T> à partir d'un T *, et respectivement un ArConstRef<T> à partir d'un const T *, il faut le faire explicitement à l'initialisation (constructeur explicit en *C++*). Des exemples sont donnés sur la figure 1.2.

```
ArRef<MyClass> r1= ... ;          // nous avons une référence sur un objet
MyClass * p1=r1.c_ptr();          // récupération du pointeur
ArRef<MyClass> r2(p1);            // initialisation explicite par le pointeur
/* r2=p1; */                      // affectation du pointeur interdite !

ArConstRef<MyClass> cr1= ... ;    // nous avons une référence sur un objet
const MyClass * cp1=cr1.c_ptr();  // récupération du pointeur
ArConstRef<MyClass> cr2(cp1);     // initialisation explicite par le pointeur
/* cr2=cp1; */                    // affectation du pointeur interdite !
```

Figure 1.2 : Passage entre T */const T * et ArRef<T>/ArConstRef<T>

Une raison qui pourrait pousser un développeur à utiliser des T */const T * à la place de ArRef<T>/ArConstRef<T> serait la mise en place d'un service passif qui désigne un ensemble d'objets uniquement à des fins statistiques par exemple. Ce service ne devrait donc en aucun cas *"référencer"* les objets, c'est à dire qu'il ne devrait pas empêcher leur destruction alors qu'ils ne sont plus référencés ailleurs dans l'application. Dans ce cas, pour éviter l'utilisation des pointeurs *C++* qui semble s'imposer ici, *ARéVi* propose les types ArPtr<T>/ArConstPtr<T> qui sont des répliques quasi-conformes de ArRef<T>/ArConstRef<T> à un détail près : ils ne *"référencent"* pas les objets désignés. Cela signifie que si un objet n'est plus désigné que par des ArPtr<T>/ArConstPtr<T>, sa destruction automatique par le ramasse-miettes a bien lieu, et les ArPtr<T>/ArConstPtr<T> qui le désignaient sont désormais automatiquement affectés à la valeur nulle. Pour revenir à notre exemple de service statistique, il suffit donc simplement de tester la nullité des ArPtr<T>/ArConstPtr<T> pour vérifier, avant de les exploiter, que les objets préalablement désignés sont bien encore présents.

L'équivalent de (T *)0 pour un ArPtr<T> est obtenu par l'appel à la méthode statique en ligne T::nullPtr(). L'équivalent de this pour un ArPtr<T> est obtenu par l'appel à la méthode en ligne thisPtr().

En résumé, nous pouvons retenir qu'une application ne doit utiliser presque essentiellement que des ArRef<T>/ArConstRef<T>. Certains services peuvent cependant nécessiter l'emploi des ArPtr<T>/ArConstPtr<T> lorsqu'ils ne doivent pas empêcher la destruction automatique des objets qu'ils utilisent. En dernier recours, l'usage des pointeurs *C++* est toujours techniquement possible, mais doit être évitée autant que faire se peut afin de limiter le risque d'effets de bords indésirables inhérent à leur utilisation.

1.1.3 Les opérateurs de conversion

En *C++*, la possibilité de convertir explicitement le type des pointeurs nous est offerte par les opérateurs `static_cast`, `dynamic_cast`, `const_cast` et `reinterpret_cast`. Puisqu'*ARéVi* décourage l'usage des pointeurs au profit des pseudo-pointeurs, des opérateurs similaires doivent être proposés.

L'opérateur `ar_down_cast` d'*ARéVi* est similaire à l'opérateur `static_cast` sur les pointeurs *C++*. L'opérateur `dynamic_cast` de *C++* n'a pas besoin d'équivalent dans *ARéVi* puisque les `ArClass` peuvent rendre un tel service (voir le paragraphe 1.2.2 de la page 10). L'opérateur `ar_const_cast` d'*ARéVi* est similaire à l'opérateur `const_cast` sur les pointeurs *C++*. L'opérateur `reinterpret_cast` de *C++* n'a pas besoin d'équivalent dans *ARéVi* puisque les pseudo-pointeurs ne concernent que les `ArObject` (`ar_down_cast` est donc suffisant). De plus, pour faciliter les conversions des vecteurs (voir le paragraphe 1.4 de la page 24) de pseudo-pointeurs, nous disposons des opérateurs `ar_down_cast_append` et `ar_const_cast_append`. La figure 1.3 illustre l'utilisation de tous ces opérateurs.

```
ArRef<BaseClass> r1= ... ;
ArRef<SubClass> r2=ar_down_cast<SubClass>(r1);

ArConstRef<ArObject> r3= ... ;
ArRef<ArObject> r4=ar_const_cast(r3);

StlVector<ArRef<BaseClass> > vr1;
// ... remplir vr1
StlVector<ArRef<SubClass> > vr2;
ar_down_cast_append<SubClass>(vr2,vr1);

StlVector<ArRef<ArObject> > vr3;
// ... remplir vr3
StlVector<ArConstRef<ArObject> > vr4;
ar_const_cast_append(vr4,vr3);
```

Figure 1.3 : Opérateurs de conversion des pseudo-pointeurs

1.1.4 Le contrôle du ramasse-miettes

Un des rôles essentiels des `ArRef<T>`/`ArConstRef<T>`, vus au paragraphe 1.1.1 de la page 2, consiste à assurer une certaine cohérence vis à vis du mécanisme de ramasse-miettes d'*ARéVi*. En conséquent, ce dernier ne s'applique qu'aux objets manipulables à travers les `ArRef<T>`/`ArConstRef<T>`, à savoir : les instances d'`ArObject` ou des classes qui en dérivent (voir le paragraphe 1.2 de la page 7).

Ce mécanisme de ramasse-miettes, bien qu'étant automatique, offre cependant des possibilités de contrôle. La possibilité de désigner un objet sans qu'il soit considéré comme *"référencé"* du point de vue du ramasse-miettes, ou encore le fait de pouvoir

```
cerr << "--- code section 1 ---" << endl;
ArRef<ArObject> r1=ArObject::NEW();       // creation d'un objet
cerr << "nb refs: "                       // il est référencé une fois
     << r1->getNbReferences() << endl;
cerr << (r1->isTransient()                // il est bien transitoire
         ? "transient"
         : "not transient") << endl;

ArPtr<ArObject> p1=r1;                    // désignation par un ArPtr<T>
cerr << "nb refs: "                       // il est toujours référencé une fois
     << r1->getNbReferences() << endl ;

r1.clear();                               // annulation de la référence
cerr << (p1 ? "not null"                  // l'objet a bien été détruit
            : "null" ) << endl;


cerr << "--- code section 2 ---" << endl;
ArRef<ArObject> r2=ArObject::NEW();       // creation d'un objet
cerr << "nb refs: "                       // il est référencé une fois
     << r2->getNbReferences() << endl;
r2->setTransient(false);                  // il devient non-transitoire

ArPtr<ArObject> p2=r2;                    // désignation par un ArPtr<T>
cerr << "nb refs: "                       // il est toujours référencé une fois
     << r2->getNbReferences() << endl;

r2.clear();                               // annulation de la référence
cerr << (p2 ? "not null"                  // l'objet n'a pas été détruit
            : "null" ) << endl;
cerr << "nb refs: "                       // il n'est plus référencé
     << p2->getNbReferences() << endl;

p2->setTransient(true);                   // il redevient transitoire
cerr << (p2 ? "not null"                  // l'objet a bien été détruit
            : "null" ) << endl;
```

```
--- code section 1---                     --- code section 2---
nb refs: 1                                nb refs: 1
transient                                 nb refs: 1
nb refs: 1                                not null
null                                      nb refs: 0
                                          null
```

Figure 1.4 : Contrôle de l'effet du ramasse-miettes

détruire explicitement un objet grâce à la méthode `destroy()` des pseudo-pointeurs, ont déja été présentés au paragraphe 1.1.2 de la page 4.

Il est également possible de choisir, pour des raisons liées à la nature même de l'application réalisée, quels sont les objets qui sont susceptibles d'être détruits automatiquement par le ramasse-miettes, les objets transitoires, et ceux qui ne le sont pas, les objets non-transitoires. Ainsi, un objet non-transitoire, même s'il n'est plus désigné par aucun `ArRef<T>`/`ArConstRef<T>`, continuera d'exister dans l'application. Ceci a du sens car il est toujours possible d'y accéder à nouveau à travers un `ArPtr<T>`/`ArConstPtr<T>` (voire un pointeur *C++*) ou en s'adressant à une instance d'`ArClass` (voir le paragraphe 1.2.2 de la page 10). Ce reglage peut être consulté et changé à tout moment à l'aide des méthodes `isTransient()` et `setTransient()` de l'objet concerné comme illustré sur la figure 1.4.

1.2 La hiérarchie `ArObject`

La classe `ArObject` représente l'ancêtre commun de toutes les classes de la bibliothèque *ARéVi* et des applications qui utilisent ses services. Elle propose un ensemble de fonctionnalités qui sont donc communes à tous les objets quels que soient leur rôle et leur signification. En particulier, la mise en œuvre des pseudo-pointeurs et du mécanisme de ramasse-miettes repose sur le fait que les objets manipulés sont de type `ArObject` ou d'un type qui en dérive.

Les objets de type `ArObject` ou d'un type dérivé ne peuvent donc être créés que par allocation dynamique et ne peuvent être manipulés qu'à travers les pseudo-pointeurs.

1.2.1 Définition d'une classe

La définition d'une nouvelle classe descendant d'`ArObject` est illustrée par l'exemple de la figure 1.5. Elle est dans l'ensemble très similaire à ce qui est habituellement fait en *C++* ; seules quelques règles systématiques sont à respecter.

```
// file myClass.h
#ifndef MYCLASS_H
#define MYCLASS_H 1

#include "AReVi/arObject.h"
using namespace AReVi;

class MyClass : public ArObject
{
public:
//-------- Type information ----------------------------------------------------
  AR_CLASS(MyClass)
//-------- Construction / Destruction -----------------------------------------
  AR_CONSTRUCTOR(MyClass)
  AR_CONSTRUCTOR_2(MyClass, int, anInteger, double, aReal)
//-------- Other methods ------------------------------------------------------
  virtual void setInteger(int anInteger);
  virtual int getInteger(void) const;
  virtual void setReal(double aReal);
  virtual double getReal(void) const;
protected:
  int _anInteger;
  double _aReal;
};
#endif // MYCLASS_H 1
```

Figure 1.5 : Définition d'une classe *ARéVi*

La classe *`MyClass`* définie ici hérite directement d'`ArObject`, ce qui nécessite l'inclusion du fichier `"AReVi/arObject.h"`. L'utilisation facultative de la directive `using namespace AReVi` permet d'accéder aux ressources d'*ARéVi* sans que le préfixe `AReVi::` ne soit requis.

La *macro* `AR_CLASS` reprenant le nom de la classe doit figurer dans la partie publique afin de mettre en place un certain nombre de mécanismes :

- ▷ interdire l'usage du constructeur par défaut, du constructeur par recopie et de l'opérateur d'affectation,
- ▷ préparer la réification de la classe définie,
- ▷ déclarer un destructeur virtuel protégé qu'il sera nécessaire d'implémenter,
- ▷ déclarer la méthode statique `CLASS()` donnant l'accès à l'instance de type `ArClass` associée la classe définie (voir le paragraphe 1.2.2 de la page 10),
- ▷ définir en ligne les méthodes statiques `nullRef()` et `nullPtr()`, représentant les pseudo-pointeurs nuls sur la classe définie,
- ▷ définir en ligne les méthodes `thisRef()` et `thisPtr()`, représentant les pseudo-pointeurs équivalents à `this` (version constante et non constante).

La déclaration des constructeurs publiques ne se fait pas directement comme en *C++*, il faut utiliser les *macros* `AR_CONSTRUCTOR`*`[_N]`* accompagnées du nom de la classe et d'une signature. La *macro* `AR_CONSTRUCTOR` correspond à un constructeur sans argument, `AR_CONSTRUCTOR_`*`N`* correspond à un constructeur à *N* arguments. Le type et le nom de chaque argument doivent être séparés par une virgule afin d'être exploités distinctement à l'intérieur de la *macro*. L'effet de l'utilisation d'une telle *macro* se résume à :

- ▷ déclarer un constructeur protégé correspondant à la signature spécifiée (avec un argument supplémentaire) qu'il sera nécessaire d'implémenter,
- ▷ définir en ligne la méthode statique *`MyClass`*`::NEW()` ayant la signature spécifiée.

C'est cette méthode statique qui doit être invoquée pour créer une nouvelle instance de la classe définie. En effet, elle ne retourne pas un pointeur mais un `ArRef<`*`MyClass`*`>`.

Le reste de la classe peut être défini de manière tout à fait habituelle et n'est soumis à aucune autre règle d'écriture spécifique à *ARéVi*. L'implémentation de la classe définie ici à titre d'exemple est donnée sur la figure 1.6 ci-contre.

L'utilisation de la *macro* `AR_CLASS_DEF` (ou `AR_CLASS_NOVOID_DEF`) est indispensable à la réification de la classe implémentée (voir le paragraphe 1.2.2 de la page 10). Il est nécessaire de lui transmettre le nom de la classe implémentée ainsi que le nom de la classe dont elle hérite.

La hiérarchie `ArObject` interdit volontairement l'héritage multiple afin de s'affranchir des difficultés qui sont généralement reprochées à ce procédé.

L'implémentation des constructeurs reprend les signatures définies par l'utilisation des *macros* `AR_CONSTRUCTOR`*`[_N]`* lors de la définition de la classe, si ce n'est qu'un argument supplémentaire, `ArCW & arCW`, doit toujours être passé en premier. Cet argument est fourni automatiquement par la méthode statique *`MyClass`*`::NEW()` correspondant au constructeur et doit simplement être transmis au constructeur de la classe dont on hérite. Mis à part l'ajout de cet argument, l'implémentation des construteurs est tout à fait classique.

```
// file myClass.cpp
#include "myClass.h"

AR_CLASS_DEF(MyClass,ArObject)

//-------- Construction / Destruction ---------------------------------------

MyClass::MyClass(ArCW & arCW)
: ArObject(arCW), _anInteger(0), _aReal(0.0) { }

MyClass::MyClass(ArCW & arCW, int anInteger, double aReal)
: ArObject(arCW), _anInteger(anInteger), _aReal(aReal) { }

MyClass::~MyClass(void) { }

//-------- Other methods ----------------------------------------------------

void MyClass::setInteger(int anInteger) { _anInteger=anInteger; }

int MyClass::getInteger(void) const { return(_anInteger); }

void MyClass::setReal(double aReal) { _aReal=aReal; }

double MyClass::getReal(void) const { return(_aReal); }
```

Figure 1.6 : Implémentation d'une classe *ARéVi*

La *macro* AR_CLASS_DEF utilisée ici est subordonnée au fait que la classe définie dispose d'un constructeur attendant l'unique argument ArCW & arCW (correspondant généralement à l'usage de la *macro* AR_CONSTRUCTOR). Cependant, si la classe définie ne dispose pas d'un tel constructeur, il faudra remplacer la *macro* AR_CLASS_DEF par la *macro* AR_CLASS_NOVOID_DEF qui s'utilise de la même façon. Ceci a pour effet de désactiver la possibilité de créer un nouvelle instance par l'ArClass associée à la classe définie (voir le paragraphe 1.2.2 de la page suivante).

Un destructeur a été implicitement déclaré par l'utilisation de la *macro* AR_CLASS lors de la définition de la classe ; son implémentation est donc nécessaire. Celle-ci et celles des autres méthodes n'ont aucune spécificité qui soit propre à *ARéVi*.

```
class MyAbstractClass : public ArObject
{
public:
//-------- Type information -------------------------------------------------
  AR_CLASS(MyAbstractClass)
//-------- Other methods ----------------------------------------------------
  virtual void doSomething(void)=0; // pure virtual
protected:
  MyAbstractClass(ArCW & arCW);
  MyAbstractClass(ArCW & arCW,int anInteger,double aReal);
};
```

Figure 1.7 : Définition d'une classe abstraite *ARéVi*

Lorsqu'on souhaite définir une classe abstraite, il est impossible d'utiliser les *macros* AR_CONSTRUCTOR*[_N]* . En effet, elles définissent des méthodes statiques en ligne qui créent une instance de la classe ! On se contente alors simplement de déclarer,

généralement dans la partie protégée de la classe, les constructeurs correspondant sans oublier leur argument supplémentaire `ArCW & arCW`, comme illustré sur la figure 1.7 de la page précédente. Pour la même raison, il est impossible d'utiliser la *macro* `AR_CLASS_DEF` ; on utilisera systématiquement la *macro* `AR_CLASS_NOVOID_DEF` pour les classes abstraites.

1.2.2 La réification des classes

Les *macros* `AR_CLASS`, `AR_CLASS_DEF` et `AR_CLASS_NOVOID_DEF` présentées au paragraphe 1.2.1 de la page 7 servent, entre autre, à la réification des classes. Ces *macros* fournissent notamment pour chaque classe *`MyClass`* une méthode statique *`MyClass`*`::REGISTER_CLASS()` qui, lorsqu'elle est invoquée pour la première fois, provoque la création d'une instance de type `ArClass` associée à la classe *`MyClass`*.

Pour que la classe *`MyClass`* soit utilisable, il est indispensable d'invoquer la méthode *`MyClass`*`::REGISTER_CLASS()` (ou celle d'une classe dérivée).

Cette invocation doit généralement avoir lieu à l'initialisation de l'application (voir la figure 1.10 de la page 18) et peut être largement facilitée grâce aux fonctionnalités proposées au paragraphe B.2.3 de la page 38. Le type `ArClass` hérite de `ArObject` et est défini dans `"AReVi/arClass.h"`. Toutefois, étant donné le rôle très particulier ce type dans *ARéVi*, il a été rendu impossible d'instancier explicitement un tel objet, ou d'hériter de ce type. Les instances d'`ArClass` sont donc créées automatiquement par *ARéVi* et il est possible d'y accéder de deux façons :

- ▷ utiliser la méthode statique *`MyClass`*`::CLASS()` fournie automatiquement lors de la définition de la classe *`MyClass`*,
- ▷ utiliser la méthode statique `ArClass::find("`*`MyClass`*`")`.

La première solution suppose que l'on connaisse le nom de la classe à laquelle on souhaite accéder au moment de l'écriture du programme. La seconde effectue la recherche d'une classe dont le nom n'est connu qu'au moment de l'exécution et est indiqué par une chaîne de caractères. Tous les `ArObject` disposent également d'une méthode `getClass()` qui désigne l'instance d'`ArClass` qui est associée à leur type. Ainsi, grâce à la méthode `isA()` de `ArClass`, il est possible d'obtenir un service proche du `dynamic_cast` de *C++*. Toutefois, les `ArClass` proposent des services qui vont bien au delà de ceux du *RTTI* de *C++* :

- ▷ inspection de la hierarchie de classes,
- ▷ obtention de toutes les instances d'une classe (avec/sans les classes dérivées),
- ▷ attribution d'alias aux instances, recherche par leur alias,
- ▷ création d'une nouvelle instance (sans introspection),
- ▷ introspection des méthodes (invocation) et des constantes.

La figure 1.8 ci-contre illustre l'utilisation de ces services. Certaines de ces méthodes utilisent des `StlVector` et des `StlString` qui sont de simples renommages des `std::vector` et `std::string` de la *STL* (voir le paragraphe 1.4 de la page 24).

```
ArRef<ArObject> r1= ... ;
cerr << "r1 is a "                          // affichage du type de r1
     << r1->getClass()->getName() << endl;

if(r1->getClass()==MyClass::CLASS())        // r1 est _exactement_ un MyClass
  { ... }

if(r1->getClass()->isA(MyClass::CLASS())    // r1 est un MyClass (ou dérivé)
  { ArRef<MyClass> r2=ar_down_cast<MyClass>(r1); }

ArConstRef<ArClass> c=ArClass::find("MyClass"); // chercher par le nom de classe
if(c)
  {
  ArConstRef<ArClass> parent=c->getParent(); // base directe

  StlVector<ArConstRef<ArClass> > ancestors;
  c->getAncestors(ancestors);               // bases directes et indirectes

  StlVector<ArConstRef<ArClass> > children;
  c->getChildren(children);                 // dérivées directes

  StlVector<ArConstRef<ArClass> > descendants;
  c->getDescendants(descendants);           // dérivées directes et indirectes

  cerr << c->getNbInstances(true) << endl;  // nombre de MyClass

  StlVector<ArRef<MyClass> > instances;
  c->getInstances(instances,true);          // récupérer les MyClass

  StlVector<StlString> aliases;
  c->getAliases(aliases,true);              // récupérer les alias des MyClass

  ArRef<MyClass> r3;
  c->findAlias("MyClass.8",r3);             // retrouver un objet par son alias
  if(r3)
    {
    cerr << r3->getAlias() << " found" << endl;
    if(!r3->setAlias("Another_Alias"))      // changer son alias
      {
      cerr << "alias already in use" << endl;
      }
    }

  ArRef<MyClass> r4;
  c->newInstance(r4);                       // demande d'instanciation
  if(r4) { ... }
  }
```

Figure 1.8 : Utilisation des `ArClass`

Les méthodes `getNbInstances()`, `getInstances()`, et `getAliases()` de `ArClass` utilisent toutes un argument `bool subClasses` qui précise si la collecte d'information doit avoir lieu uniquement dans la classe désignée ou se poursuivre aussi dans les classes dérivées. Notons que les méthodes `getInstances()` et `findAlias()` permettent de retrouver respectivement toutes les instances d'une classe donnée (et des classes dérivées) et l'instance correspondant à un certain alias. Bien entendu, les `ArClass` utilisent en interne des `ArPtr<ArObject>` afin de ne pas référencer les instances qu'elles désignent. Il est donc possible qu'un objet non-transitoire ne soit référencé à un instant donné par aucun `ArRef<T>` et qu'il puisse être néanmoins retrouvé plus tard en interrogeant une `ArClass` (voir le paragraphe 1.1.4 de la page 5).

L'attribution d'un alias aux `ArObject` est effectuée de manière automatique par leur `ArClass` respective. L'alias par défaut est constitué du nom de la classe, d'un point et d'un entier (automatiquement incrémenté à chaque instanciation). Ainsi, par exemple, l'invocation de la méthode `getAlias()` sur la huitième instance de *`MyClass`* à avoir été créée renvoie la chaîne "*`MyClass`*`.8`". Il est possible de changer l'alias d'un objet, à l'aide de sa méthode `setAlias()`. Dans ce cas, l'alias doit avoir la forme d'un identificateur (`[_a-zA-Z][_a-zA-Z0-9]*`) et la hiérarchie d'`ArClass` vérifie qu'aucun autre objet n'a déjà cet alias. À titre d'exemple, les instances d'`ArClass` ont toutes un alias constitué du nom de la classe représentée suivi de `__CLASS` (`"ArObject__CLASS"`, `"ArClass__CLASS"`, "*`MyClass`*`__CLASS"` ...)

Une instance d'`ArClass` permet également de créer une nouvelle instance du type qu'elle représente grace à sa méthode `newInstance()`. Pour que ce mécanisme soit disponible pour une classe donnée, il faut cependant que la *macro* `AR_CLASS_DEF` (et non `AR_CLASS_NOVOID_DEF`) est été employée pour la classe concernée (voir le paragraphe 1.2.1 de la page 7). Celle-ci suppose qu'il existe un constructeur n'attendant aucun autre argument que `ArCW & arCW` (ce dernier étant fourni automatiquement par l'`ArClass`). En effet, le prototype de la méthode `newInstance()` doit être fixé, il est donc impossible à un tel procédé de prévoir le passage d'arguments dont on ne connait à l'avance ni le nombre ni le type.

1.2.3 L'introspection des classes

Pour aller plus loin dans l'exploitation des `ArClass`, il est nécessaire d'utiliser l'outil `arstub` (voir le paragraphe B.3 de la page 41) afin d'instrumenter les classes. Dans ces conditions, une instance d'`ArClass` instrumentée donne accès à des `ArMethod` et des `ArConstant` qui représentent respectivement les méthodes et les constantes qui sont définies dans la classe concernée. Les types `ArMethod` et `ArConstant` héritent de `ArObject` et sont définis dans `"AReVi/arMethod.h"` et `"AReVi/arConstant.h"`. Toutefois, étant donné le rôle très particulier ces types dans *ARéVi*, il a été rendu impossible d'instancier explicitement de tels objets, ou d'hériter de ces types. Les méthodes `getNbMethods()`, `getMethod()`, `getNbConstants()` et `getConstant()` permettent de passer en revue les instances d'`ArMethod` et d'`ArConstant` qui sont associées à une classe particulière.

Une instance d'`ArConstant` est notamment représentée par le nom de la constante dans la classe (valeur symbolique d'une énumération ou `static const` *`type nom`*) et par une instance d'`ArValue` qui lui donne une valeur particulière. La classe `ArValue`, définie dans `"AReVi/arValue.h"`, hérite d'`ArObject` et représente une valeur quelconque qui corresponde à un type reconnu par les fonctionnalités d'introspection d'*ARéVi* (voir le paragraphe B.3.2 de la page 43). Un instance d'`ArMethod` est principalement représentée par le nom de la méthode décrite, son type de retour, les noms et types de ses arguments et des indicateurs précisant s'il s'agit d'un constructeur, d'une

méthode statique ou d'une méthode constante. Il est notamment possible d'invoquer l'exécution d'une `ArMethod` en passant des instances d'`ArValue` correctement typées à ses méthodes `invoke()`, `invokeConst()` ou `invokeStatic()`.

Les types représentés dans les `ArValue` et `ArMethod` sont des instances de la classe `ArType` définie dans `"AReVi/arType.h"`. Il a également été rendu impossible d'instancier explicitement de tels objets, ou d'hériter de cette classe. Les instances d'`ArType` distinguent principalement deux catégories :

- ▷ les types directs qui désignent types de base comme `int`, `StlString` ...
- ▷ les types indirects qui désignent un dérivé d'`ArObject` à travers d'une indirection (`ArRef<>`, `ArConstPtr<>` ...).

Elles indiquent également une dimension qui peut valoir :

- ▷ `0` : pour une valeur simple,
- ▷ `1` : pour un `StlVector<>` de valeurs,
- ▷ `2` : pour un `StlVector<StlVector<> >` de valeurs.

Une information concernant le mode de passage (par recopie, par référence ou par référence constante) est également contenue afin de préciser les modalités d'invocation des `ArMethod`.

Ces informations sur les types permettent donc de choisir les instances d'`ArMethod` à invoquer et de manipuler les valeurs contenues dans les instances d'`ArValue`. Concernant ce dernier point, la classe `ArValue` dispose de méthodes `is`*`TYPE`*`()`, `get`*`TYPE`*`()` et `access`*`TYPE()`* pour lire et affecter la valeur de chaque *`TYPE`* disponible. Il est à noter que l'invocation d'une méthode `get`*`TYPE`*`()` ou `access`*`TYPE()`* sur une instance d'`ArValue` n'ayant pas l'`ArType` correspondant au *`TYPE`* désigné provoquera un plantage franc de l'application.

La classe `ArClass` propose des méthodes facilitant la recherche des `ArMethod` et `ArConstant` :

- ▷ `findConstructor()` recherche dans la classe le constructeur correspondant aux types des instances `ArValue` passées en arguement,
- ▷ `findStaticMethod()`, `findMethod()` et `findConstMethod()` recherchent de la même façon la méthodes appropriée correspondant aux noms et arguments donnés, en remontant si nécessaire dans les classes de base,
- ▷ `findConstant()` recherche la constante de nom donné, en remontant si nécessaire dans les classes de base.

Bien que ces méthodes puissent effectuer leurs recherche dans les classes de base de la classe désignée, il est toujours possible de connaître l'origine de l'instance d'`ArMethod` ou d'`ArConstant` trouvée grâce à sa méthode `getDeclarationClass()` (voire `getDeclarationFile()` ou `getDeclarationLine()`).

1.2.4 La communication par messages

Toutes les instances d'`ArObject` (et donc de tous les types dérivés, même les plus utilitaires) disposent de leur propre boîte à messages dont elles peuvent consulter le contenu. Les messages sont matérialisés par des instances de la classe `Message`, définie dans `"AReVi/message.h"`, ou d'une classe dérivée. La communication par messages est asynchrone dans le sens où le fait d'envoyer un message à un objet ne provoque pas nécessairement un traitement immédiat sur ce dernier (par opposition aux appels de méthodes). Ce mode de communication permet d'envisager un certaine autonomie de décision pour les objets dans le sens où ceux-ci peuvent choisir de traiter ou non un message reçu, dans un délai variable et selon des critères changeants. Cette démarche évoque la notion d'*acteur* voire celle d'*agent*.

La méthodes `getNbMessages()` permet à un `ArObject` de connaître le nombre de messages qui sont en attente dans sa boîte. La méthode `getNextMessage()`, si la boîte est vide, renvoie une référence nulle, sinon extrait le prochain message et retourne une `ArConstRef<Message>` le désignant. Le message ne sera accessible qu'en consultation. La réception d'un message provoque l'émission d'un événement `ArObject::MessageEvent` depuis l'`ArObject` concerné. Ceci permet, par exemple, de relancer un traitement de consultation des message qui aurait été suspendu (voir le paragraphe 1.3.2 de la page 18).

Ref sur les callbacks !

Lors de la création d'un message, il est nécessaire de préciser son émetteur ; toutefois cette information peut être modifiée pour des raisons applicatives. Ensuite, il faut généralement spécifier le contenu du message. Il ne contient initialement qu'un champ de texte mais on aura tout intérêt à dériver la classe `Message` afin d'ajouter des informations spécifiques à l'objet de la communication. Lorsque le message contient toutes les informations à transmettre, il faut s'adresser à un service de messagerie qui se chargera d'acheminer le message vers son ou ses destinataires. Le fichier `"AReVi/message.h"` définit les classes `MessageService` et `NetworkMessageService` qui jouent ce rôle respectivement dans la cadre d'une application locale et dans le cadre d'une application distribuée en réseau. C'est à l'application de créer un (ou plusieurs) services de messagerie et de le (ou les) faire connaître aux objets susceptibles de communiquer par messages. Remarquons cependant qu'il n'est pas nécessaire de connaître *a priori* un tel service pour répondre à un message. En effet, le message reçu indique le service de messagerie qui l'a acheminé, il suffit donc d'utiliser celui-ci pour répondre. L'émetteur et le serveur de messagerie désignés par un message ne sont pas référencés.

L'acheminement des messages par les services de messagerie s'effectuent selon deux modes de communication :

- ▷ le mode *point-à-point* permet d'envoyer le message à un destinataire désigné explicitement par les méthodes `sendTo()` ou `sendToAlias()`,

▷ le mode *diffusion* permet d'envoyer le message, par la méthode `broadcast()`, à l'ensemble des objets qui se sont abonnés à ce type de message avec la méthode `subscribeBroadcast()` du service de messagerie concerné.

Les abonnés à la diffusion ne sont pas référencés par le service de messagerie.

Un message diffusé par la méthode `broadcast()` d'un service de messagerie ne peut être reçu par un `ArObject` que si ce dernier s'est abonné à ce type de message (ou à un de ses types de base) grâce à la méthode `subscribeBroadcast()` de ce même service de messagerie.

L'utilisation d'un `NetworkMessageService` doit être associée au choix d'un port *UDP* afin que les différents services distribués puissent entrer en contact. Les services distribués sur un réseau local doivent normallement se contacter automatiquement s'ils utilisent le même port. Si ce n'est pas le cas, ou si les ports sont différents, ou encore si l'on souhaite sortir du réseau local, les méthodes `tryToReachLocalSessions()` et `tryToReachSession()` permettent de réitérer les tentatives de contact. Les services contactés sont obtenus par la méthode `getDistantSessions()`. Il faut également invoquer régulièrement la méthode `watchNetwork()` de chaque `NetworkMessageService` afin que les tentatives de contact et les échanges de messages soient pris en compte ; ceci fait généralement l'objet d'une activité (voir le paragraphe 1.3.2 de la page 18).

```
ArRef<MessageService> svc=NetworkMessageService::NEW(1234);
ArRef<Message> msg=Message::NEW(thisRef());
msg->setText("Is there anybody out there ?");
svc->broadcast(msg);
```

```
ArRef<MessageService> svc=NetworkMessageService::NEW(1234);
svc->subscribeBroadcast(Message::CLASS(), // recevoir les ‘‘Message’’ qui
                        thisRef());       // sont diffusés

...

ArRef<Message> msg=getNextMessage();      // consulter les messages
if(msg)                                   // et repondre
  {
  cerr << msg->getText() << endl;
  ArRef<Message> reply=Message::NEW(thisRef());
  reply->setText("I'm not sleeping");
  msg->getService()->sendToAlias(msg->getEmitterAlias(),reply);
  }
```

Figure 1.9 : Communication par messages

Dans le cadre d'une application distribuée en réseau, les messages sont *sérialisés* par le service expéditeur, réinstanciés (à l'aide de leur `ArClass`) et *désérialisés* par les services récepteurs avant d'être déposés dans la boîte des destinataires. Ceci suppose que les types de messages utilisés proposent un constructeur avec l'argument unique `ArCW & arCW` (voir le paragraphe 1.2.2 de la page 10) et qu'ils implémentent leurs méthodes `writeToStream()` et `readFromStream()` (voir le paragraphe 1.2.5 de la page suivante). Ceci implique également que les références sur l'expéditeur et sur les destinataires ne sont utilisables que dans le cadre d'un échange local. Pour exploiter la

communication en réseau, il est donc nécessaire de ne reposer que sur l'utilisation des alias (`getEmitterAlias()`, `sendToAlias()` ...).

Chaque `NetworkMessageService` dispose d'un identifiant de session qui sert à compléter automatiquement les alias des objets concernés par les échanges de messages afin de différencier les objets ayant le même alias sur des sessions différentes. Ainsi, par exemple, si l'objet ayant pour alias `"ArObject.2"` diffuse une message via le service ayant l'identifiant de session `"hostname:1234/35462"`, ce message, lorsqu'il sera reçu, donnera la réponse `"ArObject.2@hostname:1234/35462"` pour sa méthode `getEmitterAlias()`. Les objets récepteurs pourront alors répondre à l'émetteur en passant cette chaîne à la méthode `sendToAlias()` du service de messagerie. Le fait que cette chaîne soit *"lisible"* est tout à fait anecdotique, comme l'illustre la figure 1.9 de la page précédente dans laquelle aucun alias n'apparaît explicitement. Notons que si on utilise la méthode `sendToAlias()` avec un alias non suivi d'un identifiant de session, alors on s'adresse nécessairement à un destinataire de la session locale. La méthode `sendToSameAlias()` envoie le message à l'objet qui a l'alias spécifié sur chaque session en contact.

1.2.5 Divers services d'`ArObject`

La sérialisation/désérialisation et été évoquée à propos de l'acheminement des messages au paragraphe 1.2.4 de la page 14. Tout objet est potentiellement en mesure d'inscrire ses propriétés dans un flux de sortie et de les extraire depuis un flux d'entrée grâce aux méthodes `writeToStream()` et `readFromStream()` de la classe `ArObject`. Par défaut celles-ci ne font rien et c'est donc aux classes applicatives de les surdéfinir pour effectuer les opérations attendues (c'est ce que fait la classe `Message` et ce que doivent faire les classes dérivées).

Ref sur les flux !

Le mécanisme de traitement des *exceptions* de type `try/catch/throw` n'est pas utilisé dans *ARéVi* simplement par choix personnel concernant le style d'écriture. En effet il provoque des branchements extra-procéduraux, assimilables à des sauts non-locaux de type `setjmp()/longjmp()`, qui sont encore plus difficile à suivre que les `goto` tant décriés. L'idée initiale consistant à apporter une réaction appropriée pour chaque cause particulière de l'echec d'un traitement est dans la pratique très difficile à mettre en œuvre. Dans la pratique on constate que l'utilisation de ce mécanisme en est très loin et que, dans le meilleur des cas, on se contente d'abandonner le traitement qui échoue (pour une raison dont on ne se soucie guère) afin de laisser l'application se poursuivre, et le plus souvent on se contente d'imprimer un message d'erreur et de mettre fin à l'application. Ainsi, l'utilisation d'un mécanisme aussi couteux, uniquement pour ces deux types de réactions, n'a pas été retenu dans *ARéVi*.

En règle générale, les méthodes d'*ARéVi*, lorsqu'elles sont susceptibles d'échouer, indiquent par leur valeur de retour si le traitement s'est déroulé correctement. Il suffit

alors de tester de manière claire et explicite ce résultat. Le seul point génant concerne les constructeurs des objets ; en effet, si un objet est confronté à une erreur durant sa construction, il est tout de même instancié (on n'obtient pas de référence nulle). Il est donc possible de demander à un objet s'il est dans un état incorrect grâce à sa méthode `fail()`. Celle-ci renvoie `true` si un appel à `setErrorMessage()` avec une chaîne non vide a été effectué sur cet objet (généralement lors de sa construction). La méthode `getErrorMessage()` renvoie alors le message en question. Si on sait remédier au problème, on peut invoquer à nouveau la méthode `setErrorMessage()` de l'objet concerné avec une chaîne vide après l'avoir *"réparé"* ; sa méthode `fail()` renvoie désormais `false`.

Les instances d'`ArObject` permettent d'embarquer un pointeur à usage *a priori* indéterminé (`void *`) grâce aux méthodes `setData()` et `getData()`. Cette possibilité s'adresse aux programmeurs qui souhaitent intégrer *ARéVi* avec des bibliothèques ou des applications existantes et ne concerne pas les utilisateurs principaux d'*ARéVi*. Ce pointeur permet de confier à un `ArObject` des données fournies par la bibliothèque ou l'application associée et de les récupérer plus tard afin de les exploiter avec la bibliothèque ou l'application en question.

1.3 Le singleton `ArSystem`

La classe `ArSystem`, définie dans `"AReVi/arSystem.h"` ne s'inscrit pas du tout dans la hiérarchie des `ArObject`. Elle sert principalement à initialiser l'application et à effectuer une boucle de simulation qui active les différents traitements tout en assurant la mise à jour de l'affichage et la réaction aux interventions de l'utilisateur. Accessoirement, cette classe encapsule des services de programmation système. Toutes les méthodes de la classe `ArSystem` sont statiques.

1.3.1 L'initialisation de l'application

L'instantiation d'un `ArSystem` assure l'initialisation des services d'*ARéVi* alors que sa destruction rend ces services inutilisables. Bien entendu, cette instance doit être unique. Généralement, cette instanciation a lieu dans le programme principal (la fonction `main()`) de l'application. Le constructeur attend les arguments de la ligne de commande afin de les mémoriser pour pouvoir les restituer plus tard dans l'application. Un argument optionnel permet de préciser que l'on ne souhaite pas utiliser les services relatifs à la *3D*. Il est toutefois possible d'effectuer ces mêmes opérations d'initialisation et de destruction à l'aide des méthodes statiques `ArSystem::init()` et `ArSystem::destroy()`. Ceci peut être utile pour embarquer *ARéVi* dans une autre application (voir l'annexe C de la page 45).

```
#include "AReVi/arSystem.h"
// ...
int main(int argc,char ** argv)
{
ArSystem arevi(argc,argv);                      // initialisation d'AReVi
MyClass1::REGISTER_CLASS();                     // initialisation des classes
MyClass2::REGISTER_CLASS();

if(!ArSystem::loadPlugin("Imlib2ImageLoader"))  // si plugin non disponible
  {
  ArSystem::loadPlugin("MagickImageLoader");    // essayer celui-ci
  }

StlVector<StlString> opts;
opts.push_back("an option");
opts.push_back("another one");                  // charger un plugin
ArSystem::loadPlugin("MyPlugin",&opts);         // attendant des options

// ...                                          // boucle de simulation
return(0);
}
```

Figure 1.10 : Initialisation d'*ARéVi*

Après l'initialisation d'*ARéVi*, et des classes (voir le paragraphe 1.2.2 de la page 10), il est possible de charger des *plugins*, en leur passant éventuellement des arguments, comme illustré sur la figure 1.10. Ces *plugins* permettent de fournir à *ARéVi* des fonctionnalités issues d'autres bibliothèques sans pour autant en être dépendant. Ainsi, si les ressources dont dépend un *plugin* ne sont pas disponibles, cela n'empêche pas le reste de l'application de fonctionner. Les *plugins* suivants sont disponibles avec *ARéVi* :

- ▷ `MagickImageLoader` (repose sur *ImageMagick*) : charger des images (textures),
- ▷ `Imlib2ImageLoader` (repose sur *Imlib2*) : charger également des images,
- ▷ `AviRecorder` (repose sur *FFmpeg*) : générer des vidéos à partir des vues *3D*,
- ▷ `SoundESD` (repose sur *ESD*) : capturer/jouer du son `3D`,
- ▷ `Sound3DPlayer` (repose sur *OpenAL*) : jouer du son `3D`,
- ▷ `CgWrapper` (repose sur *Cg*) : utiliser les *vertex/pixel shaders*,
- ▷ `XmlParser` (repose sur *LibXml2*) : utiliser des flux *XML*,
- ▷ `Text2Speech` (repose sur *ElanSpeech*) : produire une synthèse vocale,
- ▷ `JpegLib` (repose sur *JpegLib*) : compresser/décompresser des images,
- ▷ `ZLib` (repose sur *ZLib*) : compresser/décompresser des données.

Bien entendu, chacun est libre de développer et d'ajouter de nouveaux *plugins* (voir le paragraphe B.4 de la page 44).

1.3.2 La boucle de simulation

Le déroulement normal d'une application *ARéVi* se décompose généralement en deux phases bien distinctes :

- ▷ l'initialisation de la simulation par la création d'un ensemble d'objets dotés d'une autonomie d'exécution,
- ▷ la simulation proprement dite qui consiste à laisser *"vivre"* ces objets.

De nouveaux objets peuvent bien entendu apparaître en cours de simulation. De plus, pendant toute la durée de la simulation, *ARéVi* doit se charger de mettre à jour les vues *3D* et de détecter les actions de l'utilisateur (clavier, souris ...). L'activation des entités autonomes est assurée par un ordonnanceur qui gère des activités. Ces notions sont représentées respectivement par les classes `Scheduler` (`"AReVi/scheduler.h"`) et `Activity` (`"AReVi/activity.h"`) ; ce sont des `ArObject`.

L'ordonnanceur est un singleton, il peut être obtenu par les méthodes statiques `Scheduler::get()` et `Scheduler::access()`. Il est responsable de l'avancement du temps et doit donc exécuter les activités lorsque leur date de déclenchement respective est atteinte. À un instant donné, il sélectionne toutes les activités qui ont une date de déclenchement antérieure ou égale et provoque leur exécution.

Durant l'exécution de toutes les activités retenues à un instant donné, le temps est considéré comme indiscernable ; la méthode `getTime()` de l'ordonnanceur renvoie la même valeur pendant tout le cycle d'activation.

Cependant, pour des raisons utilitaires, extérieures au principe même de la simulation réalisée, comme des acquisitions/emissions de données externes, les activités peuvent être réparties selon trois priorités :

- ▷ `0`, priorité *"normale"* (par défaut),
- ▷ `1`, activation en début de cycle,
- ▷ `-1`, activation en fin de cycle.

En revanche, pour une priorité donnée et un instant donné, il n'y a aucune raison que les activités s'exécutent selon un ordre préétabli.

Afin de minimiser le risque de biais dans une simulation, les activités ayant la même priorité sont exécutées dans un ordre aléatoire au sein de chaque cycle d'activation.

L'instant courant est déterminé par l'ordonnanceur et est exprimé sous la forme d'un réel (`double`) que l'on peut obtenir par sa méthode `getTime()`. Tous les calculs sur le temps sont donc sujets à des erreurs d'arrondis, c'est pourquoi il est nécessaire de fournir une précision à la construction de l'ordonnanceur. Tous les calculs sur le temps sont arrondis à cette précision qui doit être négligeable devant le plus petit ordre de grandeur temporel de l'application. Cette opération d'arrondi est accessible par la méthode `roundTime()` de l'ordonnanceur. Dans ces conditions, le test d'égalité stricte entre deux instants a du sens.

La classe `Scheduler` est une classe abstraite et *ARéVi* en propose deux implémentations qui se distinguent par la manière de déterminer l'instant courant :

- ▷ `RealTimeScheduler` considère que le temps s'écoule comme à notre montre,
- ▷ `VirtualTimeScheduler` considère que le temps *"saute"* d'instant remarquable en instant remarquable.

Dans tous les cas, il est possible de suspendre/relancer l'ordonnanceur à l'aide de sa méthode `setSuspended()`. Bien entendu, pendant la durée de suspension, l'ordonnanceur considère que le temps ne s'est pas écoulé.

Pour l'ordonnanceur *"temps réel"* (`RealTimeScheduler`), l'heure courante est demandée au système d'exploitation à chaque début de cycle d'activation. Cette valeur représente le nombre de secondes depuis le démarrage de la simulation. Il est possible d'accélérer ou de ralentir le temps perçu par un facteur multiplicatif réglé à l'aide de sa méthode `setTimeFactor()`. Un cycle d'activation du `RealTimeScheduler` concerne donc toutes les activités qui ont une date de déclenchement antérieure ou égale à la date déterminée. Ce déclenchement peut éventuellement avoir un retard non négligeable par rapport à la date prévue si la machine est extrèmement chargée en calcul. Ce type d'ordonnanceur est particulièrement adapté aux application de simulation interactive puisque la notion de temps utilisée dans la simulation correspond à la notion de temps perçue par l'utilisateur.

L'ordonnanceur *"temps virtuel"* (`VirtualTimeScheduler`), quant à lui, détermine l'heure courante en début de cycle comme étant la date de déclenchement prévue la plus proche. Ainsi, toutes les activités dont la date de déclenchement prévue est strictement égale à cet instant sont exécutées. Quel que soit le temps de calcul nécessaire aux différents cycles d'activation, nous sommes sûrs que le temps de simulation ne prendra jamais de retard ; toutefois ce temps de simulation n'a plus rien à voir avec le temps perçu par l'utilisateur et est exprimé dans une unité qui ne doit pas nécessairement être assimilé à des secondes (on peut l'interpréter comme des picosecondes ou des millénaires). Ce type d'ordonnanceur est particulièrement adapté aux simulations dans lequels le temps doit être controlé avec une échelle très fine au détriment de l'interactivité avec l'utilisateur.

Une activité doit être créée en précisant un délai de déclenchement ; c'est à dire que la date de son premier déclenchement correspond à l'instant courant de l'ordonnanceur plus le délai spécifié. L'exécution d'une activité consiste simplement à appeler une méthode sur un `ArObject`. Cette méthode doit avoir le prototype suivant :

- ▷ `bool` *`myMethod`*`(ArRef<Activity> act,double dt);`

et doit être affecté, ainsi que l'objet concerné, à l'activité à l'aide de sa méthode `setBehavior()`. Ainsi, très souvent les objets créent eux-mêmes les activités qui les concernent comme illustré sur la figure 1.11 ci-contre. Bien entendu, il est possible de créer plusieurs activités pour un même objet, celles-ci sont tout à fait indépendantes.

Une activité maintient une référence sur l'objet qui lui a été confié par `setBehavior()` ; un objet doté d'une activité ne disparaît donc pas automatiquement, il est *"maintenu en vie"* pour un traitement ultérieur.

```
// ...

MyClass::MyClass(ArCw & arCW)
: ArObject(arCW)
{
ArRef<Activity> act=Activity::NEW(0.1); // déclencher dans 0.1 unités de temps
act->setBehavior(thisRef(),&MyClass::myBehavior);
}

bool // re-activate
MyClass::myBehavior(ArRef<Activity> act,
                    double dt)
{
cerr << getAlias() << "/" << act->getAlias() << " dt=" << dt << endl;
return(true); // nouvelle activation dans 0.1 unités de temps
}

// ...
```

Figure 1.11 : Création d'une activité

La méthode appelée par le déclenchement doit renvoyer un `bool` indiquant si l'activité doit être relancée une prochaine fois. Si cette valeur vaut `true`, le délai passé au constructeur est réutilisé pour déterminer la date du prochain déclenchement. Ce délai, également assimilé à une période, peut être modifié à tout moment avec la méthode `setInterval()` de l'activité. De la même façon, la priorité (`0`, `1` ou `-1`) d'une activité peut être ajustée avec sa méthode `setPriority()`. Une activité peut donc servir à provoquer un traitement unique dans le futur ou bien un traitement répétitif en jouant simplement sur la valeur de retour du traitement en question. On peut même imaginer un traitement répétitif qui s'arrête à un moment donné (il renvoie `false`) sur un critère applicatif (l'objectif de l'activité a été atteint).

Cette même méthode reçoit en argument l'activité qui est en train de l'exécuter ainsi que la durée qui s'est écoulée (toujours du point de vue de l'ordonnanceur) depuis le précédent déclenchement, ou depuis l'instantiation de l'activité pour le premier déclenchement. Avec un ordonnanceur *"temps virtuel"*, l'argument `dt` correspond exactement à la période de l'activité. En revanche, avec une ordonnanceur *"temps réel"*, il peut y avoir du retard et `dt` est nécessairement supérieur ou égal à la période spécifiée. Il est par exemple envisageable, dans une simulation interactive, de comparer l'argument `dt` avec la période de l'activité (obtenu avec sa méthode `getInterval()`) et ajuster celle-ci quand le retard est manifestement trop important (la machine est trop chargée en calcul). Notons qu'à l'inverse, une activité peut recevoir une période nulle (dans son constructeur ou avec `setInterval()`), ce qui signifie qu'elle sera déclenchée aussi souvent que possible, c'est à dire à chaque cycle de l'ordonnanceur.

Une activité est dans l'état *terminé* (méthode `isEnded()`) lorsque la méthode qu'elle exécute renvoie `false` ou bien lorsqu'on invoque sa méthode `abort()` ; elle

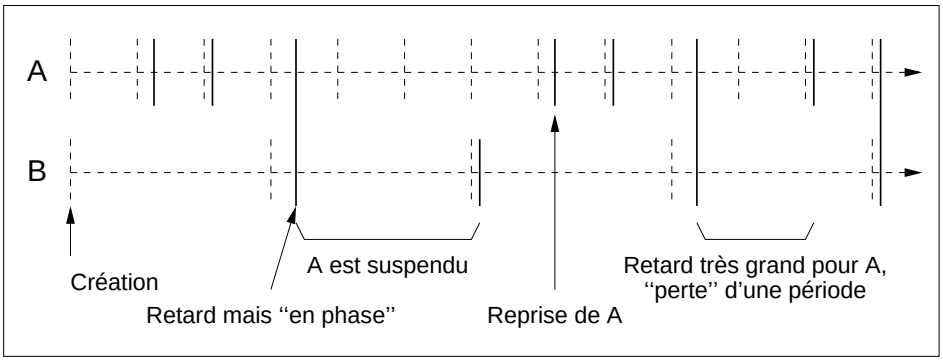


Figure 1.12 : Déclenchement des activités

devient alors inutilisable. Il est également possible de suspendre et relancer une activité à l'aide de sa méthode `setSuspended()`. L'exemple de la figure 1.12 illustre le fait que, même avec un ordonnanceur *"temps réel"*, les retards ne s'accumulent pas et les activités qui sont prévues pour être *"en phase"* le restent. Ici l'activité `B` doit se déclencher tous les trois déclenchements de `A`. En effet, le calcul de la prochaine date de déclenchement d'une activité ne dépend pas de la date effective du déclenchement courant mais de sa date prévue. Ce principe est respecté également pour la relance d'une activité suspendue ; elle reprend sur un multiple de sa période. Cependant, comme indiqué en fin d'exemple, si le retard d'une activité dépasse sa période, des déclenchements sont perdus ; l'argument `dt` indique toutefois de manière correcte le temps écoulé depuis le déclenchement précédent.

```
#include "AReVi/arSystem.h"

ArRef<Scheduler> myInitFunc(void)
{
ArRef<Scheduler> sched=RealTimeScheduler::NEW(1e-6); // choix d'un ordonnanceur
// ArRef<Scheduler> sched=VirtualTimeScheduler::NEW(1e-6);

// ... créer des objets ayant des activités ...

return(sched);
}

int main(int argc,char ** argv)
{
ArSystem arevi(argc,argv);             // initialisation
// ... initialisation des classes, chargement des plugins ...
ArSystem::simulationLoop(&myInitFunc); // boucle de simulation
return(0);
}
```

Figure 1.13 : Lancement de la simulation

Le choix d'un ordonnanceur et la boucle d'activation doivent se faire comme indiqué sur la figure 1.13. Il faut fournir une fonction sans arguments renvoyant un ordonnanceur. Cette fonction commence par instancier un ordonnanceur *"temps réel"* ou *"temps virtuel"* qui sera renvoyé en fin de fonction. Il n'y a plus alors qu'à instancier des objets qui auront des activités afin qu'ils puissent faire *"vivre"* la simulation. Cette fonc-

tion est transmise par pointeur à la méthode statique `ArSystem::simulationLoop()` qui l'invoque une fois pour obtenir l'ordonnanceur. Cette même méthode effectue alors une boucle qui consiste à activer l'ordonnanceur et à gérer l'affichage et les interactions avec l'utilisateur. L'invocation de la méthode statique `ArSystem::leaveSimulation()` durant la simulation, provoque la sortie de la boucle de simulation. De plus, si lors de l'initialisation d'*ARéVi*, l'argument optionnel indique que l'on ne souhaite pas utiliser les services relatifs à la *3D* (voir le paragraphe 1.3.1 de la page 17), alors la boucle de simulation se terminera automatiquement dès qu'il n'y aura plus d'activités à exécuter.

```
int main(int argc,char ** argv)
{
// ... initialiser la bibliothèque ...
ArSystem arevi(argc,argv);              // initialisation d'ARéVi
// ... initialisation des classes, chargement des plugins ...
if(ArSystem::enterSimulation(&myInitFunc))
  {
  while(ArSystem::simulationStep())
    {
    // ... traitements NON-BLOQUANTS de la bibliothèque ...
    }
  }
return(0);
}
```

Figure 1.14 : Ouverture de la boucle de simulation

Cette boucle de simulation peut être ouverte afin de pouvoir y insérer des traitements supplémentaires. Cette solution est particulièrement utile lorsqu'on souhaite utiliser une bibliothèque fournissant des services tels que des interfaces graphiques. Dans ce cas, comme illustré sur la figure 1.14, on initialise la bibliothèque en question au début du programme, puis on initialise *ARéVi* comme à l'accoutumée, et on remplace l'appel à la méthode statique `ArSystem::simulationLoop()` par une boucle explicite. On débute la simulation par l'appel de la méthode statique `ArSystem::enterSimulation()` qui renvoie `true` si la simulation peut continuer. Ensuite, chaque invocation de la méthode statique `ArSystem::simulationStep()` représente un cycle d'activation de l'ordonnanceur et permet de gérer l'affichage et les interactions avec l'utilisateur. Si cette invocation renvoie `true` la simulation ne doit pas se terminer ; on doit donc poursuivre la boucle de simulation. Le corps de cette boucle peut alors contenir des traitements permettant à la bibliothèque utilisée de jouer son rôle (prendre en compte les événements graphiques par exemple). Il est impératif que ces traitements soient non-bloquants sans quoi la simulation resterait figée en attendant les circonstances susceptibles de provoquer le débloquage.

1.3.3 Fonctionnalités utilitaires

XXX idx XXX `kw` XXX `kwi1` XXX

1.4 Quelques types utilitaires

Chapitre 2

Second chapitre

Blabla blabla.

2.1 Première section

Blabla blabla.

2.1.1 Première sous-section

Blabla blabla.

2.1.2 Seconde sous-section

Blabla blabla.

2.2 Seconde section

Blabla blabla.

2.2.1 Première sous-section

Blabla blabla.

2.2.2 Seconde sous-section

Blabla blabla.

Annexe A

Construire et installer *ARéVi*

Ce chapitre indique, sans entrer dans les moindres détails, la démarche à suivre pour compiler la bibliotheque *ARéVi* et l'installer. Pour plus d'information sur ce qui est effectué durant ce procédé, il suffit de consulter l'annexe B de la page 33 ; en effet la construction d'*ARéVi* repose exactement sur les mêmes principes que la construction d'une application ou d'une bibliothèque utilisant *ARéVi*.

Pour des raisons de portabilité, les commandes indiquées ici n'utilisent aucune option qui soit spécifique à un environnement particulier. Ces commandes fonctionnent donc avec l'ensemble des systèmes sur lesquels *ARéVi* est supporté.

A.1 Installer les archives

La première étape consiste en la récupération de l'archive contenant le code source d'*ARéVi* et éventuellement de celle contenant les programmes de tests. Ces archives sont disponibles sur la page `http://www.enib.fr/~harrouet/` et sont nommées de la façon suivante :

- ▷ `AReVi_`*YYYY*`m`*MM*`d`*DD*`.tgz` pour *ARéVi*
- ▷ `AReVi_`*YYYY*`m`*MM*`d`*DD*`_tests.tgz` pour les programmes de tests

Les motifs *YYYY*, *MM* et *DD* donnés ici sont des caractères numériques qui représentent respectivement l'année, le mois et le jour de mise à disposition de cette version d'*ARéVi*. Le numéro de version est obtenu en combinant ces trois informations en un seul entier.

- ▷ ex : `AReVi_2006m08d02.tgz` pour la version `20060802` du 02 août 2006.

Les programmes de tests ont été placés dans une archive séparée car il ne sont pas indispensables pour l'utilisation d'*ARéVi* et sont accompagnés d'un grand volume de données.

Une fois ces archives obtenues et placées dans un répertoire de travail de notre choix, il suffit de les décompresser de la manière suivante :

- ▷ `gzip -cd AReVi_`*YYYY*`m`*MM*`d`*DD*`.tgz | tar xvf -`
- ▷ `gzip -cd AReVi_`*YYYY*`m`*MM*`d`*DD*`_tests.tgz | tar xvf -`

Ceci a pour effet de créer un répertoire `AReVi_`*YYYY*`m`*MM*`d`*DD* dans lequel se trouvent toutes les informations fournies par les archives. Nous trouvons notamment dans ce répertoire un fichier `README` qui donne un résumé des opérations de construction et d'installation.

A.2 Compiler *ARéVi*

ARéVi est principalement constitué d'une bibliothèque contenant la très grande majorité des services proposés. Cette bibliothèque est accompagnée de *plugins* fournissant des services annexes. Un fichier *script* `armake` est fourni pour automatiser la construction de l'ensemble. Pour l'utiliser il suffit de suivre les étapes suivantes :

- ▷ se placer dans le répertoire obtenu précédemment,
  `cd AReVi_`*YYYY*`m`*MM*`d`*DD*
- ▷ générer les fichiers `makefile` pour la bibliothèque *ARéVi* et les *plugins*,
  `./armake conf`
- ▷ compiler la bibliothèque *ARéVi* et les *plugins*.
  `./armake`

Le *script* `armake` comprend d'autres options dont on peut obtenir la liste à l'aide de la commande `./armake help`. Notamment il peut construire la bibliothèque de distribution *ARéViNS* avec la commande `./armake ns` et les programmes utilitaires (dans le sous-répertoire `Tools`) avec la commande `./armake tools`. Il peut également construire les programmes de tests (dans le sous-répertoire `Tests`) avec la commande `./armake tests`.

La compilation de la bibliothèque *ARéVi* doit se dérouler sans difficulté. En revanche la compilation des *plugins* est subordonnée à l'existence et au bon fonctionnement d'autres outils ou bibliothèques optionnels qui justifie leur mise à disposition sous forme de *plugin*. Il se peut alors que, durant la compilation de certains de ces *plugins*, on obtienne un affichage similaire à celui-ci :

```
!!! Dependency problem concerning the following file(s):
!!!     text2Speech.cpp
!!!     text2SpeechSync.cpp
!!!     text2SpeechASync.cpp
!!!
!!! You can stay in this situation if you think you don't need
!!!     ../../../lib/libAReViText2Speech.so
!!! or you will have to find and install the required tools/libraries.
```

Comme l'indique ce message, cette situation n'est probablement pas grave. Si l'utilisation du *plugin* en question n'est pas envisagée on peut laisser cette situation en l'état. En revanche, si on tient absolument à l'utiliser, il est alors nécessaire d'installer correctement sur la machine l'outil ou la bibliothèque qui est requis pour la réalisation du *plugin* incriminé. Les fichiers sources de ces *plugins* doivent normallement contenir des commentaires indiquant le nom de l'outil requis ainsi que son site *internet*. Il reste alors à suivre les instructions d'installation spécifiques à l'outil en question.

A.3 Installer *ARéVi*

Le *script* `armake` permet également d'installer *ARéVi* automatiquement dans le répertoire `/usr/local`. Naturellement, cette opération ne peut être effectuée que par l'administrateur du système (`root`). Cette installation est déclenchée par la commande `./armake install` et effectue les opérations suivantes :

- ▷ effacer, s'il existe, le répertoire `/usr/local/AReVi_`*YYYY*`m`*MM*`d`*DD*,
- ▷ effacer, s'il existe, le répertoire `/usr/local/AReVi`,
- ▷ effacer, s'il existe, le fichier `/usr/local/bin/arevi-config`,
- ▷ créer le répertoire `/usr/local/AReVi_`*YYYY*`m`*MM*`d`*DD* et y placer les fichiers qui sont nécessaire à l'utilisation d'*ARéVi* (en-têtes, bibliothèques ...),
- ▷ créer le lien symbolique `/usr/local/AReVi` afin qu'il désigne le répertoire `/usr/local/AReVi_`*YYYY*`m`*MM*`d`*DD*,
- ▷ créer le lien symbolique `/usr/local/bin/arevi-config` désignant le fichier *script* `/usr/local/AReVi/lib/arevi-config`.

Ainsi, la version d'*ARéVi* nouvellement installée est directement accessible dans `/usr/local/AReVi` et le *script* `arevi-config` nécessaire à l'utilisation d'*ARéVi* (voir l'annexe B de la page 33) est directement accessible parmis les autres commandes du système déja placées en `/usr/local/bin`.

Bien entendu, si nous n'avons pas les droits pour administrer le système il est tout à fait envisageable d'installer *ARéVi* dans un autre répertoire de notre choix. C'est d'ailleurs ce que propose la commande `./armake install` lorsqu'elle detecte que nous ne sommes pas `root` ; elle invite à effectuer les opérations suivantes :

- ▷ créer un répertoire `AReVi_`*YYYY*`m`*MM*`d`*DD* dans le répertoire d'installation,
- ▷ recopier, dans le répertoire `AReVi_`*YYYY*`m`*MM*`d`*DD* nouvellement créé, les répertoires `AReVi_`*YYYY*`m`*MM*`d`*DD*`/include` et `AReVi_`*YYYY*`m`*MM*`d`*DD*`/lib`,
- ▷ s'assurer que le *script* `AReVi_`*YYYY*`m`*MM*`d`*DD*`/lib/arevi-config` est bien accessible par la variable d'environnement `PATH`.

Il est important de noter que le répertoire contenant *ARéVi* doit contenir dans son nom le motif `AReVi_`*YYYY*`m`*MM*`d`*DD*. En effet, le numéro de version d'*ARéVi* n'est pas noté *"en dur"* mais est déterminé automatiquement depuis ce motif lors de l'invocation du *script* `arevi-config`.

A.4 Spécificités de certains systèmes

ARéVi n'a été développé et testé que sur les quelques systèmes dont nous pouvons disposer, à savoir :

- ▷ *Linux/ix86/x86-64/PPC*
- ▷ *Window$/Cygwin*
- ▷ *MacOsX 10.4*
- ▷ *IRIX 6.5*
- ▷ *FreeBSD 5.2*

Il est toutefois extrêmement probable qu'*ARéVi* puisse fonctionner sous d'autres systèmes. Il faut dans ce cas compléter le *script* `arevi-config`[@] (voir l'annexe B de la page 33) afin qu'il prenne en compte les spécificités du nouveau système.

La construction, l'installation et l'utilisation d'*ARéVi* ne reposent que sur des outils et des commandes standard (`flex`, `bison`, `sh`, `make`, `sed`, `awk` ...) en veillant à ne pas utiliser d'options qui soient spécifiques à un système particulier. Il est indispensable de s'assurer de la disponibilité de ces outils.

Comme évoqué au paragraphe A.2 de la page 28, la compilation d'un *plugin* peut échouer à cause de l'absence de l'outil ou de la bibliothèque que celui-ci exploite. Il n'est cependant pas toujours nécessaire d'obtenir l'outil en question depuis son site principal. En effet, il est souvent plus simple de vérifier s'il n'en existe pas un paquetage prêt à l'emploi pour le système utilisé (distribution *Linux*, *Cygwin* ...). Il est à noter toutefois que certains de ces outils ne fonctionnent pas sur tous les systèmes ; dans ces conditions, le *plugin* *ARéVi* correspondant ne sera, bien évidemment, pas disponible pour ce système.

Concernant le rendu *3D*, *ARéVi* utilise *OpenGL* dans sa version 1.1 afin de limiter autant que faire se peut la dépendance à des fonctionnalités *3D* avancées qui ne sont pas nécessairement disponibles sur toutes les plateformes. Toutefois, *ARéVi* utilise le mécanisme d'extensions d'*OpenGL* afin de bénéficier de ces fonctionnalités lorsqu'elles sont disponibles. Il est donc problable que sur certaines plateformes, des messages semblables à celui-ci apparaissent durant l'initialisation :

```
!!! GL_EXT_separate_specular_color extension is not supported !!!
```

Ceci n'a, bien évidemment, aucune conséquence sur les services qui ne dépendent pas de la fonctionnalité particulière indiquée.

Si les applications *AReVi* ne trouvent pas les bibliothèques dynamiques dont elles ont besoin à l'exécution, il peut être nécessaire, sur certaines plateformes, de compléter la variable d'environnement `LD_LIBRARY_PATH` avec le chemin `/usr/local/AReVi/lib` (ou le répertoire d'installation approprié).

[@] Merci de m'en faire part.

A.4.1 Spécificités de *MacOsX*

Étant donné que *MacOsX* (à partir de la version 10.3) est livré en standard avec un serveur *X*, la partie graphique d'*ARéVi* n'a pas pas été réécrite de manière spécifique à cette plateforme ; il est donc nécessaire de lancer le serveur *X* pour exécuter les applications *ARéVi*.

Pour que les applications *ARéVi* puissent trouver les bibliothèques dynamiques dont elles ont besoin à l'exécution, il est nécessaire de compléter la variable d'environnement `DYLD_LIBRARY_PATH` avec le chemin `/usr/local/AReVi/lib` (ou le répertoire d'installation approprié).

A.4.2 Spécificités de *Window$*

Le système *Window$* est assez particulier dans le sens où il ne suit pas les recommandations *POSIX* et ne repose en général que sur des standards qui lui sont propres et incompatibles avec le reste du parc informatique. Pour assurer cependant la portabilité du code et des outils, il suffit simplement de recourir à l'environnement *Cygwin*[Ⓑ]. La partie graphique d'*ARéVi* pour *Window$* a entièrement été réécrite en *Win32* ; il n'est donc nullement nécessaire d'utiliser un serveur *X* sur cette plateforme.

Pour que les applications *ARéVi* puissent trouver les bibliothèques dynamiques dont elles ont besoin à l'exécution, il est nécessaire de compléter la variable `PATH` avec le chemin `/usr/local/AReVi/lib` (ou le répertoire d'installation approprié). Une autre solution moins élégante mais habituelle sur ce système consiste en la recopie de toutes les bibliothèques dynamiques nécessaires dans le même répertoire que le programme exécutable.

Pour qu'une application compilée sur un poste *Window$/Cygwin* puisse être installée sur un autre poste *Window$*, il est nécessaire d'effectuer une installation minimale de *Cygwin* car celle-ci positionne correctement la base de registres. Il est tout à fait envisageable de supprimer le répertoire `Cygwin` créé par cette opération, mais il ne faut en aucun cas retirer les informations qui ont été placées dans la base de registres. Sans cette installation, l'application pourra donner l'illusion de fonctionner mais aura un comportement instable. Bien entendu, il faudra que le fichier exécutable de l'application soit accompagné de toutes les bibliothèques dynamiques requises. Pour faciliter cette recherche, la commande :

▷ `cygcheck` *`exe_ou_dll`*

permet de lister toutes les bibliothèques dynamiques qui sont liées à l'exécutable ou à la bibliothèque dynamique qui lui est indiqué en paramètre.

Ⓑ `http://www.cygwin.com/`

Annexe B

Développer avec *ARéVi*

Bien que ce ne soit pas rigoureusement indispensable, le développement avec *ARéVi* suppose l'utilisation du *script* `arevi-config` fourni lors de l'installation d'une part, et la rédaction d'un *script* `configure` qui permet de faciliter la génération d'un fichier `makefile` standard d'autre part. Ces *scripts* se contentent d'utiliser les outils et commandes standards évoqués au paragraphe A.4 de la page 30. Ils ont pour intérêts principaux la prise en charge de la portabilité du développement et l'automatisation des traitements de compilation. Ainsi, si les consignes données ici sont respectées, il n'y a rien à faire pour porter un développement en *ARéVi* sur différents systèmes.

B.1 Utiliser le *script* `arevi-config`

Le propos de ce *script* est de fournir les paramètres qui sont spécifiques à la plateforme utilisée en ce qui concerne le développement avec *ARéVi*. L'invocation de ce *script* provoque simplement l'écriture sur la sortie standard d'une réponse aux requêtes passées sur sa ligne de commande. Ce résultat peut donc être directement exploité par d'autres fichiers *scripts* ou `makefile`. Dans la pratique, l'utilisateur n'invoque que très peu ce *script* ; ce sont les *scripts* `configure` qui en font le plus grand usage.

B.1.1 Informations d'ordre général

Le nom du répertoire d'installation d'*ARéVi* peut être obtenu en invoquant la commande `arevi-config --install`. Ceci donne généralement un résultat semblable à `/usr/local/AReVi_`*YYYY*`m`*MM*`d`*DD*. Cette même information permet de déterminer le numéro de version (comme indiqué au paragraphe A.1 de la page 27) ; c'est le propos de la commande `arevi-config --version`.

L'invocation de `arevi-config --sys` provoque simplement l'affichage du nom du système utilisé (`Linux_ppc` pour un *PowerPC* sous *Linux*, `Darwin` pour un *Apple* sous *MacOsX* ...). Ceci permet de faire des choix dépendant de la plateforme bien que dans la pratique ce ne soit que très rarement nécessaire. À ce propos, la commande `arevi-config --libvar` retourne le nom de la variable d'environnement qui permet l'accès aux bibliothèques dynamiques du système :

- ▷ `LD_LIBRARY_PATH` sous *UNIX* en général,
- ▷ `DYLD_LIBRARY_PATH` sous *MacOsX*,
- ▷ `PATH` sous *Window$*
- ▷ ...

Les conventions de nommage des bibliothèques ne sont d'ailleurs pas les mêmes sur tous les systèmes. C'est pourquoi les commandes `arevi-config --libformat` *NAME* et `arevi-config --slibformat` *NAME* donnent respectivement le nom de bibliothèque dynamique et statique correspondant à la base *NAME* sur le système utilisé :

- ▷ `lib`*NAME*`.so` et `lib`*NAME*`.a` sous *UNIX* en général,
- ▷ `lib`*NAME*`.dylib` et `lib`*NAME*`.a` sous *MacOsX*,
- ▷ *NAME*`.dll` et *NAME*`.a` sous *Window$*,
- ▷ ...

B.1.2 Options de compilation

Pour effectuer la compilation proprement dite, il est nécessaire de déterminer le nom du compilateur *C++* disponible ; c'est le rôle de la commande `arevi-config --cc`.

Le fichier *script* `configure` effectue un calcul automatique des dépendances entre fichiers (voir le paragraphe B.2.2 de la page 38). Pour ceci, il est nécessaire de déterminer l'option du compilateur qui permet cette opération ; celle-ci est indiquée par le résultat de la commande `arevi-config --deps`.

La compilation des fichiers sources nécessite généralement des options (optimisations, inclusions, *macros* ...). La commande `arevi-config --cflags` les fournit pour une compilation optimisée. Pour remplacer les options d'optimisation par celles de *débugage* il suffit d'ajouter `--debug` à cette commande. Ces options de compilation contiennent notamment la définition de la *macro* `AR_VERSION` indiquant le numéro de version d'*ARéVi* tel que présenté au paragraphe A.1 de la page 27. L'édition de liens nécessite également l'utilisation d'options (répertoires, bibliothèques ...). Celles-ci sont données par la commande `arevi-config --ldflags`[@]. L'ajout de l'option `--ns` permet d'utiliser la bibliotheque de distribution *ARéViNS*. Les options d'édition de liens vues ici s'appliquent aussi bien à la réalisation de programmes exécutables qu'à celle de bibliothèques dynamiques. Cependant, il existe des options, parfois com-

[@] L'option `--self` évite la liaison de la bibliothèque *ARéVi* avec elle-même lors de sa construction

plexes, qui sont spécifiques à la réalisation d'un de ces types de fichiers binaires. Le propos des commandes `arevi-config --exec` *`NAME`*, `arevi-config --dynlib` *`NAME`* et `arevi-config --plugin` *`NAME`* est donc de fournir la commande permettant de réaliser respectivement le programme exécutable, la bibliothèque dynamique ou le *plugin* de nom *`NAME`*. Bien entendu, la commande obtenue doit être complétée par la liste des fichiers objets à lier et les options d'édition de liens évoquées précédemment.

Pour utiliser le compilateur *C++* proposé par *Intel*[β], l'option `--intel` doit être ajoutée à toutes les commandes `arevi-config` évoquées dans ce paragraphe (B.1.2).

Enfin, l'option `--jni` permet de réaliser un *plugin* pour l'environnement *Java* (voir le paragraphe C.2 de la page 46). À cette fin, il est nécessaire de l'ajouter aux commandes `arevi-config` qui utilisent les options `--cflags`, `--ldflags` et `--libformat` *`NAME`*[γ] lors de la réalisation de ce *plugin* pour *Java*.

B.2 Rédiger et utiliser un *script* `configure`

La réalisation de programmes exécutables ou de bibliothèques nécessite généralement la rédaction d'un fichier `makefile`. Toutefois, selon les circonstances (plateforme utilisée, disponibilité d'un outil ...) ce fichier doit prendre différentes formes. Le programme `make` de certaines plateformes[δ] peut proposer des extensions qui permettent d'exprimer ces variantes mais ce n'est pas le cas pour la version standard de cet outil. De plus, certains traitements assez systématiques, tel le calcul des dépendances entre fichiers, devraient être recopiés quasi-intégralement dans les fichiers `makefile` de chaque réalisation.

Le propos du fichier *script* `configure` est donc de prendre en compte les différentes variantes possibles et de rédiger le fichier `makefile` standard approprié. Ce fichier rédigé par l'utilisateur pour chacune de ses réalisation doit être succint, facile à rédiger et à modifier. Il consiste simplement en le renseignement de quelques variables qui sont exploitées par un *script* annexe, tout en laissant à l'utilisateur l'opportunité d'insérer des règles de son choix dans le fichier `makefile` généré. Les *scripts* mis en œuvre par ce procédé sont portables et ne reposent que sur les outils et commandes standards évoqués au paragraphe A.4 de la page 30[ε].

[β] Disponible sur `http://www.intel.com/software/products/compilers/clin/`

[γ] *Java* sous *MacOsX* impose que ses *plugins* aient l'extension `.jnilib` et non l'extension `.dylib` qui désigne habituellement les bibliothèques dynamiques et les *plugins* sur cette plateforme !

[δ] C'est le cas de *GNU-make* par exemple, qui permet d'exprimer des alternatives.

[ε] Pour toutes ces raisons, nous n'avons pas recours aux complexes outils *GNU-autoconf-automake-libtool* qui nécessitent notamment une installation (dans la bonne version) sur les plateformes cibles.

B.2.1 Trame principale d'un fichier `configure`

Pour présenter simplement les étapes de la rédaction d'un tel fichier, nous nous réfererons à l'exemple de la figure B.1. Celui-ci permet la réalisation de deux programmes exécutables `myExecA` et `myExecB` à partir des fichiers sources `myExecA1.cpp`, `myExecA2.cpp`, `myExecB1.cpp` et `myExecB2.cpp`.

```
#!/bin/sh
#-------- Global settings ------------------------------------------------------
ARCONFIG=`which arevi-config`
if [ ! -x "${ARCONFIG}" ] ; then
  echo "Cannot find arevi-config !"
  exit 1
fi

ARDIR="`${ARCONFIG} --install`"
ARLIB="${ARDIR}/lib"

CCFLAGS="`${ARCONFIG} --cflags`"
LDFLAGS="`${ARCONFIG} --ldflags`"

LIB_HINT="`${ARCONFIG} --libvar`"
LIB_HINT="export ${LIB_HINT}=\\\$\$${LIB_HINT}:${ARLIB}:."

#-------- Executable settings --------------------------------------------------

EXEC_TARGET="myExecA myExecB"
myExecA_FILES="myExecA1.cpp myExecA2.cpp"
myExecB_FILES="myExecB1.cpp myExecB2.cpp"

#-------------------------------------------------------------------------------
# Generate 'makefile' according to the above settings

set LIB_HINT # make some variables visible in the makefile
. ${ARLIB}/build-makefile > makefile

#-------- Inserted in makefile -------------------------------------------------
# Specific targets can be resolved before/after 'all' or 'clean' are called

post_all :
        @echo ; echo "${LIB_HINT}" ; echo

#-------------------------------------------------------------------------------
```

Figure B.1 : Exemple de fichier `configure`

La première ligne rappelle qu'il s'agit d'un fichier script tout à fait habituel. Il faudra veiller à le rendre exécutable (par la commande `chmod +x ./configure`) afin de pouvoir l'utiliser.

La première étape de la rédaction nécessite le renseignement de la variable `ARCONFIG` en s'assurant qu'elle permet bien d'invoquer le *script* `arevi-config` (voir le paragraphe B.1 de la page 33). Ceci permet notamment de renseigner les variables `ARDIR` (voir le paragraphe B.1.1 de la page 33), `CCFLAGS` et `LDFLAGS` en invoquant cette commande (voir le paragraphe B.1.2 de la page 34). Les variables `CCFLAGS` et `LDFLAGS` peuvent également être complétées par d'autres options nécessaire à cette réalisation.

La seconde étape consiste simplement en l'énumération des cibles à réaliser et celle de leurs fichiers sources respectifs. La variable `EXEC_TARGET` indique donc les noms des programmes exécutables à réaliser. Ensuite, pour chaque cible *name*, il faut renseigner une variable *name*`_FILES` qui énumère l'ensemble des fichiers sources à compiler pour l'obtenir.

À ce stade de la rédaction du fichier `configure` l'utilisateur a indiqué les informations essentielles à la réalisation des cibles souhaitées. Celles-ci sont alors exploitées par le *script* `build-makefile` (fourni lors de l'installation d'*ARéVi*) dont le résultat constitue le contenu du fichier `makefile`. Ce script exploite le contenu des variables `ARCONFIG`, `CCFLAGS`, `LDFLAGS`, `EXEC_TARGET` et *name*`_FILES` vues ici, ainsi que quelques autres présentées au paragraphe B.2.3 de la page suivante. Le fichier `makefile` généré propose notamment les cibles `all` (implicite) pour l'ensemble de notre réalisation et `clean` pour effacer tous les fichiers qui peuvent être regénérés par ce procédé.

L'invocation du *script* `build-makefile` a lieu *"en ligne"* grâce à la commande `.` (invocation dans le *shell* courant). Celui-ci se termine par un `exit`, ce qui implique que toutes les lignes du fichier `configure` qui suivent l'invocation de `build-makefile` ne sont pas exécutées ; au contraire, elles sont insérées telles quelles dans le fichier `makefile` par le *script* `build-makefile`. En particulier, si les règles `pre_all`, `post_all`, `pre_clean` ou `post_clean` sont trouvées, alors elles seront invoquées automatiquement en début/fin d'invocation de `all`/`clean`. Ceci permet d'exprimer dans le fichier `makefile` des traitements qui ne concernent pas directement *ARéVi*.

De plus, `build-makefile` effectue le calcul de dépendances sur les fichiers sources qui lui sont indiqués. Si lors de ce calcul un fichier source n'est pas trouvé, il essaye de le produire en invoquant les règles de `makefile` fournies. C'est une situation assez courante d'avoir à compiler un fichier source qui est généré par un outil spécifique ; c'est le cas des *parsers* avec des outils comme `flex` et `bison`, ou encore des *"méta-fichiers"* avec le préprocesseur `moc` de la bibliothèque graphique *Qt*.

Pour que les règles de `makefile` indiquées à la fin du fichier `configure` puissent utiliser les variables initialisées dans ce script, il est nécessaire qu'elles soient recréées et initialisées à l'intérieur du `makefile` généré. La commande `set` qui précède l'invocation du script `build-makefile` sert donc à indiquer à ce script les variables concernées par cette opération.

L'exemple de règle de `makefile` donné dans la figure B.1 ci-contre permet simplement, après la réalisation des cibles principales (`post_all`), d'afficher les commandes que l'utilisateur peut avoir à saisir pour que les bibliothèques soient trouvées à l'exécution (voir les paragraphes A.4.1 et A.4.2 de la page 31). Cette règle affiche le contenu de la variable `LIB_HINT`© dont la valeur a été déterminée dans le *script* `configure`, c'est pourquoi celle-ci a été passée à la commande `set` juste avant l'invocation du *script* `build-makefile`.

© Celle-ci n'a rien d'obligatoire, il s'agit d'une indication donnée à l'utilisateur.

B.2.2 Invocation du *script* `configure`

Pour réaliser les cibles décrites dans le fichier `configure`, il faut commencer par invoquer ce *script* par la commande `./configure`. Ceci a pour effet de calculer les dépendances des fichiers sources tout en rédigeant un fichier `makefile` standard. Il ne reste plus alors qu'à invoquer la commande `make` afin de réaliser les cibles attendues. L'exemple de la figure B.1 de la page 36 est utilisé sur la figure B.2.

```
$ ./configure
======== Building dependencies for myExecA ========
myExecA1.cpp
myExecA2.cpp
======== Building dependencies for myExecB ========
myExecB1.cpp
myExecB2.cpp
$
$ make

export DYLD_LIBRARY_PATH=$DYLD_LIBRARY_PATH:/usr/local/AReVi_2006m08d02/lib:.

$
```

Figure B.2 : Invocation du *script* `configure`

Bien entendu, à chaque modification des fichiers sources il est uniquement nécessaire de relancer la commande `make`. Toutefois, si de nouveaux fichiers sont ajoutés, ou simplement si de nouvelles directives `#include` sont utilisées, il est nécessaire de regénérer les dépendances en invoquant `./configure` à nouveau.

La commande `make clean` efface tous les fichiers qui sont produits automatiquement ; le fichier `makefile` lui-même est donc supprimé par cette opération puisqu'il sera généré à nouveau lors d'une future invocation du *script* `configure`.

B.2.3 Les variables du fichier `configure`

Le paragraphe B.1 de la page 36 indique le principe général de la rédaction d'un fichier `configure` et en présente les principales variables. Cependant, il existe d'autres variables significatives pour ce *script* qui permettent de réaliser d'autres types de cibles et d'exprimer des réglages plus fins.

Concernant les cibles disponibles, nous avons vu que la variable `EXEC_TARGET` désignait les programmes exécutables. De la même façon, les variables `PLUGIN_TARGET`, `DYNLIB_TARGET` et `STATLIB_TARGET` désignent respectivement les cibles qui sont des *plugins* et des bibliothèques dynamiques ou statiques. Leur utilisation suit la même logique que pour celle de `EXEC_TARGET`, seulement nous avons vu au paragraphe B.1.1 de la page 33 que les conventions de nommage des bibliothèques pouvaient varier d'un système à un autre. Nous évitons donc de mettre dans ces variables les noms

complets des bibliothèques à réaliser ; nous leur préférons leur nom de base `name` et nous indiquons, grâce à la variable ***name*`_FULLNAME`**, le nom complet qu'il faudra utiliser lors de la réalisation de la cible, comme indiqué sur la figure B.3. L'utilisation de la variable ***name*`_FULLNAME`** est envisageable également pour les cibles de `EXEC_TARGET` bien que cela ne soit que plus rarement nécessaire.

```
# ...
PLUGIN_TARGET="myPluginA myPluginB"

myPluginA_FULLNAME="../MyLibDir/`${ARCONFIG} --libformat myPluginA`"
myPluginA_FILES="myA1.cpp myA2.cpp"

myPluginB_FULLNAME="../MyLibDir/`${ARCONFIG} --libformat myPluginB`"
myPluginB_FILES="myPluginB1.cpp myPluginB2.cpp"

DYNLIB_TARGET="myDynlibA myDynlibB"

myDynlibA_FULLNAME="../MyLibDir/`${ARCONFIG} --libformat myDynlibA`"
myDynlibA_FILES="myDynlibA1.cpp myDynlibA2.cpp"

myDynlibB_FULLNAME="../MyLibDir/`${ARCONFIG} --libformat myDynlibB`"
myDynlibB_FILES="myDynlibB1.cpp myDynlibB2.cpp"

STATLIB_TARGET="myStatlibA myStatlibB"

myStatlibA_FULLNAME="../MyLibDir/`${ARCONFIG} --slibformat myStatlibA`"
myStatlibA_FILES="myStatlibA1.cpp myStatlibA2.cpp"

myStatlibB_FULLNAME="../MyLibDir/`${ARCONFIG} --slibformat myStatlibB`"
myStatlibB_FILES="myStatlibB1.cpp myStatlibB2.cpp"
# ...
```

Figure B.3 : Réaliser des *plugins* et des bibliothèques avec `configure`

Les options de compilation et d'édition de liens sont données d'une manière globale par les variables `CCFLAGS` et `LDFLAGS`. Cependant, il peut être nécessaire de les différencier selon les cibles à réaliser. Ceci peut être effectué à plusieurs niveaux :

- pour l'ensembles des fichiers exécutables, des *plugins*,
  des bibliothèques dynamiques ou statiques,
  `EXEC_CCFLAGS="${CCFLAGS} ... more options ..."`
  `EXEC_LDFLAGS="${LDFLAGS} ... more options ..."`
  `PLUGIN_CCFLAGS="${CCFLAGS} ... more options ..."`
  `PLUGIN_LDFLAGS="${LDFLAGS} ... more options ..."`
  `DYNLIB_CCFLAGS="${CCFLAGS} ... more options ..."`
  `DYNLIB_LDFLAGS="${LDFLAGS} ... more options ..."`
  `STATLIB_CCFLAGS="${CCFLAGS} ... more options ..."`
  `STATLIB_LDFLAGS="${LDFLAGS} ... more options ..."`
- pour une cible ***name*** particulière.
  ***name*`_CCFLAGS="${CCFLAGS} ... more options ..."`**
  ***name*`_LDFLAGS="${LDFLAGS} ... more options ..."`**

Il reste à voir une dernière variable du fichier `configure` qui propose une facilité permettant d'automatiser l'invocation des méthodes ***MyClass*`::REGISTER_CLASS()`** évoquées au paragraphe 1.2.2 de la page 10. En effet, pour chaque classe ***MyClass***

dérivée d'`ArObject` constituant l'application *name*, il est nécessaire d'invoquer la méthode correspondante lors de l'initialisation de l'application. La variable *name*`_CLASSES` permet de repérer toutes les classes des fichiers désignés par *name*`_FILES` et de générer un fichier source supplémentaire contenant une unique fonction qui effectue toutes les invocations de *MyClass*`::REGISTER_CLASS()`. Il suffit alors de déclarer et d'invoquer cette unique fonction à l'initialisation de l'application.

```
# ...

EXEC_TARGET="myExec"
myExec_FILES="myExec.cpp myClassA.cpp myClassB.cpp"
myExec_CLASSES="${ARLIB}/register-classes myClasses.cpp \
                MyNameSpace registerMyClasses"

# ...
```

```
/****
This file is generated by the register-classes script
Do not edit or your changes will be lost !
****/

namespace MyNameSpace {

class MyClassA { public: static void REGISTER_CLASS(void); };
class MyClassB { public: static void REGISTER_CLASS(void); };

void
registerMyClasses(void)
{
MyClassA::REGISTER_CLASS();
MyClassB::REGISTER_CLASS();
}

} // namespace MyNameSpace
```

Figure B.4 : Automatiser l'enregistrement des `ArClass`

Sur l'exemple de la figure B.4 nous cherchons à réaliser un programme exécutable `myExec` à partir du programme principal `myExec.cpp` et de l'implémentation des deux classes `MyClassA` et `MyClassB` dans les fichiers `myClassA.cpp` et `myClassB.cpp`. La variable *myExec*`_CLASSES` indique qu'il faut invoquer le *script* `register-classes` (fourni lors de l'installation d'*ARéVi*) afin qu'il produise un fichier `myClasses.cpp` définissant la fonction `registerMyClasses()`. Les fichiers indiqués dans *myExec*`_FILES` sont inspectés[η] en supposant que les classes rencontrées, et la fonction `registerMyClasses()` générée sont dans l'espace de nommage `MyNameSpace` (indiquer `::` si aucun espace de nommage n'est utilisé). La figure B.4 indique à titre illustratif le fichier généré suite à l'utilisation d'une variable *name*`_CLASSES`. Ce fichier est bien entendu compilé et lié avec les fichiers indiqués dans *name*`_FILES`. La fonction générée devra donc être déclarée et invoquée à l'initialisation de l'application. Ce procédé s'applique aussi à la réalisation de *plugins* ; dans ce cas l'invocation de la fonction générée doit avoir lieu à l'initialisation du *plugin* (voir le paragraphe B.4 de la page 44).

[η] À la recherche des `AR_CLASS`*`[_NOVOID]`*`_DEF`, voir le paragraphe 1.2.1 de la page 7.

B.3 Instrumenter le code avec arstub

aujouter ref

Les instances d'`ArClass` donnent accès aux objets de type `ArMethod` et `ArConstant` qui les caractérisent. Bien entendu, ces objets n'apparaissent pas spontanément ; ils sont générés grâce à l'invocation de l'outil `artool` fourni avec *ARéVi*. Cet outil analyse les fichiers d'entête (`.h`) des classes soumises à l'introspection et génere le code d'une fonction dont l'invocation attribuera leurs `ArMethod` et `ArConstant` aux `ArClass` à instrumenter.

B.3.1 Utilisation de l'outil arstub

L'outil `arstub` est construit depuis le répertoire `Tools/StubGenerator` d'*ARéVi* et est installé dans le répertoire `lib` (avec `arevi-config`). Pour fonctionner, il utilise l'outil *Doxygen*[ϑ] qui doit être correctement installé. Ce dernier permet de générer des fichiers *XML* qui décrivent le contenu des classes analysées ; il est donc nécessaire de s'assurer que le *plugin* *XmlParser* d'*ARéVi* est correctement construit puisque l'outil `arstub` l'utilise afin d'extraire le contenu des fichiers *XML* générés par *Doxygen*. L'outil `arstub` termine son travail en générant un ensemble de fichiers *C++* qu'il suffit de compiler et lier avec les autres fichiers de code source de l'application réalisée.

L'invocation de cet outil s'accompagne du passage de quelques options sur la ligne de commande :

- ▷ `-d` *`doxygen_command`* : indique la commande pour invoquer l'outil *Doxygen* ; par défaut `doxygen` est utilisé,
- ▷ `-f` *`stub_function_name`* : indique le nom de la fonction qui sera générée ; cet argument est obligatoire,
- ▷ `-o` *`output_dir`* : indique le nom du répertoire dans lequel vont être générés les fichiers *XML* et *C++* ; cet argument est obligatoire, le répertoire est vidé de son contenu au lancement de l'outil,
- ▷ `-i` *`include_path`* : indique le nom d'un répertoire (ou d'un fichier) qui contient les entêtes à analyser ; cet argument est obligatoire et peut être utilisé plusieurs fois pour indiquer plusieurs emplacements,
- ▷ `-s` *`strip_include_path`* : indique le préfixe à retirer des chemins précisés par `-i` pour écrire les directives `#include` (ex : `-i ../../include/AReVi` et `-s ../../include` → `#include "AReVi/..."`) ; cet argument est optionnel et peut être utilisé plusieurs fois,
- ▷ `-e` *`enum_name`* : indique qu'un type *`enum_name`* inconnu de l'outil est une énumération et doit être traité comme un `int` ; cet argument est optionnel et peut être utilisé plusieurs fois.

[ϑ] `http://www.doxygen.org`

```
# ...

##------- Executable settings -----------------------------------------------
DOXYCMD="`which doxygen`"
ARSTUB="${ARLIB}/arstub"
if [ ! -x "${DOXYCMD}" ] ; then
  echo "Cannot find doxygen ! (no stub generation)"
elif [ ! -x "${ARSTUB}" ] ; then
  echo "Cannot find ${ARSTUB} ! (no stub generation)"
else
  LIBVAR="`${ARCONFIG} --libvar`"
  eval ${LIBVAR}=\$${LIBVAR}:${ARLIB} ${ARSTUB} \
       -d "${DOXYCMD}" -f initMyClassesReflection -o StubDir \
       -i "../include/MyClasses" -s "../include" -e "ForeignClass::AnEnum"
fi

EXEC_TARGET="myApp"
myApp_FILES="myApp.cpp ../src/MyClasses/*.cpp StubDir/*.cpp"

# ...
```

Figure B.5 : Générer l'instrumentation des classes

La figure B.5 illustre l'utilisation de l'outil `arstub` depuis un *script* `configure`. Nous réalisons ici un exécutable `myApp` qui principalement constitué du programme `myApp.cpp` et des classes implémentées dans le répertoire `../src/MyClasses`. Nous souhaitons instrumenter le code de ces classes, c'est pourquoi nous ajoutons à la liste des fichiers sources le contenu du répertoire `StubDir`. Ces derniers fichiers sont obtenus par l'invocation de l'outil `arstub`. Nous vérifions toutefois que cet outil ainsi que *Doxygen* sont accessibles. L'outil `arstub` utilise la bibliothèque *AReVi*, il est donc nécessaire de s'assurer qu'il trouvera cette bibliotheque quelle que soit la plateforme de développement utilisée. À cet effet, nous recueillons le nom de la variable d'environnement appropriée (dans `LIBVAR`, voir le paragraphe B.1.1 de la page 33) et ajustons cette variable dans la commande `eval` qui invoque `arstub`. Nous retrouvons ici l'usage des options décrites précédement ; la fonction générée sera donc :

- ▷ `void initMyClassesReflection(void)`

dans le répertoire `StubDir`. Les entêtes des classes à instrumenter sont situées dans le répertoire `../include/MyClasses` mais les directives `#include` doivent omettre le préfixe `../include`. Enfin, le type `ForeignClass::AnEnum` représente une énumération qui n'est pas décrite dans l'ensemble des classes analysées ; ce type sera alors assimilé à `int` dans l'instrumentation générée.

Bien entendu, les fichiers sources générés peuvent être utilisés pour constituer une bibliothèque ou un *plugin*. L'instrumentation des classes de la bibliothèque *AReVi* est d'ailleurs réalisée sous la forme d'un *plugin* nommé `CoreReflection`. Quelle que soit la manière d'embarquer dans l'application le code généré par `arstub`, l'important est d'appeler la fonction indiquée par l'option `-f` (il faudra la déclarer, comme pour l'usage du *script* `register-classes` vu au paragraphe B.2.3 de la page 38) afin d'attribuer aux `ArClass` à instrumenter leurs objets `ArMethod` et `ArConstant`. Comme dans le cas des méthodes *`MyClass`*`::REGISTER_CLASS()` (voir le paragraphe 1.3.1 de la page 17), la fonction `main()` ou (l'initialisation d'un *plugin*) est un bon endroit pour cet appel.

B.3.2 Analyse du code par l'outil `arstub`

Lorsqu'on invoque l'outil `arstub`, celui-ci analyse tous les fichiers d'entête qu'il trouve à partir des emplacements désignés par ses options `-i`. L'analyse ne concerne que les classes qui s'inscrivent dans la hiérarchie d'`ArObject`, c'est à dire les classes qui utilisent la *macro* `AR_CLASS` (voir le paragraphe 1.2.1 de la page 7). En effet, seules ces classes sont associées à une instance d'*ArClass* et les foncctionnalités d'introspection sont justement proposées par les `ArClass`. Si une *macro* `AR_CLASS` est précédée de la *macro* `AR_UNSTUBBED`, alors cette classe sera ignorée par l'analyse. Ceci peut se justifier pour certaines classes utilitaires d'assez bas niveau n'ayant d'intérêt que dans la contexte de *C++*.

Parmi chaque classe retenue, l'analyse ne concerne que ce qui est publique. Les méthodes sont analysées, y compris les `AR_CONSTRUCTOR`*`[_N]`* et les méthodes statiques. Les types de retours et d'arguments de ces méthodes sont analysés et doivent appartenir à l'ensemble suivant :

- ▷ `bool`, `float`, `double` et `StlString`,
- ▷ `char`, `short`, `int` et `long long`, ainsi que leur version `unsigned`,
- ▷ `ArRef<>`, `ArConstRef<>`, `ArPtr<>` et `ArConstPtr<>`,
- ▷ `StlVector<>` et `StlVector<StlVector<> >` des types précédents.

Les passages ou retours de ces types par référence et référence constante sont également reconnus. Les types qui ne sont pas reconnus sont recherchés parmis les énumérations des classes analysées et en dernier recours parmis les options `-e` de l'invocation d'`arstub`. Si le type initialement inconnu est trouvé lors de cette recherche il sera alors assimilé à un `int` dans l'instrumentation qui sera générée. En revanche, si une méthode utilise un type de retour ou un argument inconnu, celle-ci est rejetée avec un message d'avertissement. C'est le cas par exemple pour des méthodes qui utilisent des pointeurs ; celles-ci ne peuvent être utilisée que depuis *C++*. Dans ce cas il est possible d'utiliser à nouveau la *macro* `AR_UNSTUBBED` au début de la déclaration de la méthode afin qu'elle soit silencieusement ignorée. Les méthodes `template` sont également rejettées. Chaque méthode non rejettée sera alors réprésentée par une instance d'`ArMethod` dans l'`ArClass` concernée.

L'analyse de chaque classe retenue concerne également ses énumérations et ses constantes (`static const` *`type nom`*`;`). Chaque valeur d'une énumération sera représentée dans l'`ArClass` concernée par une instance d'`ArConstant` considérée comme un `int` ayant cette valeur. Les autres constantes sont sujettes aux mêmes contraintes que celles vues pour les types des méthodes. L'utilisation de la *macro* `AR_UNSTUBBED` au début de la déclaration d'une telle constante permet une nouvelle fois d'ignorer silencieusement les types inconnus. Chaque constante non rejettée sera représentée dans l'`ArClass` concernée par une instance d'`ArConstant` ayant le type et la valeur en question.

B.4 Réaliser un *plugin* ARéVi

Blabla blabla.

B.4.1 Première sous-section

Blabla blabla.

B.4.2 Seconde sous-section

Blabla blabla.

Annexe C

Embarquer *ARéVi*

Blabla blabla.

C.1 Embarquer *ARéVi* dans *Tcl/Tk*

Blabla blabla.

C.1.1 Première sous-section

Blabla blabla.

C.1.2 Seconde sous-section

Blabla blabla.

C.2 Embarquer *ARéVi* dans *Java*

Blabla blabla.

C.2.1 Première sous-section

Blabla blabla.

C.2.2 Seconde sous-section

Blabla blabla.

Annexe D

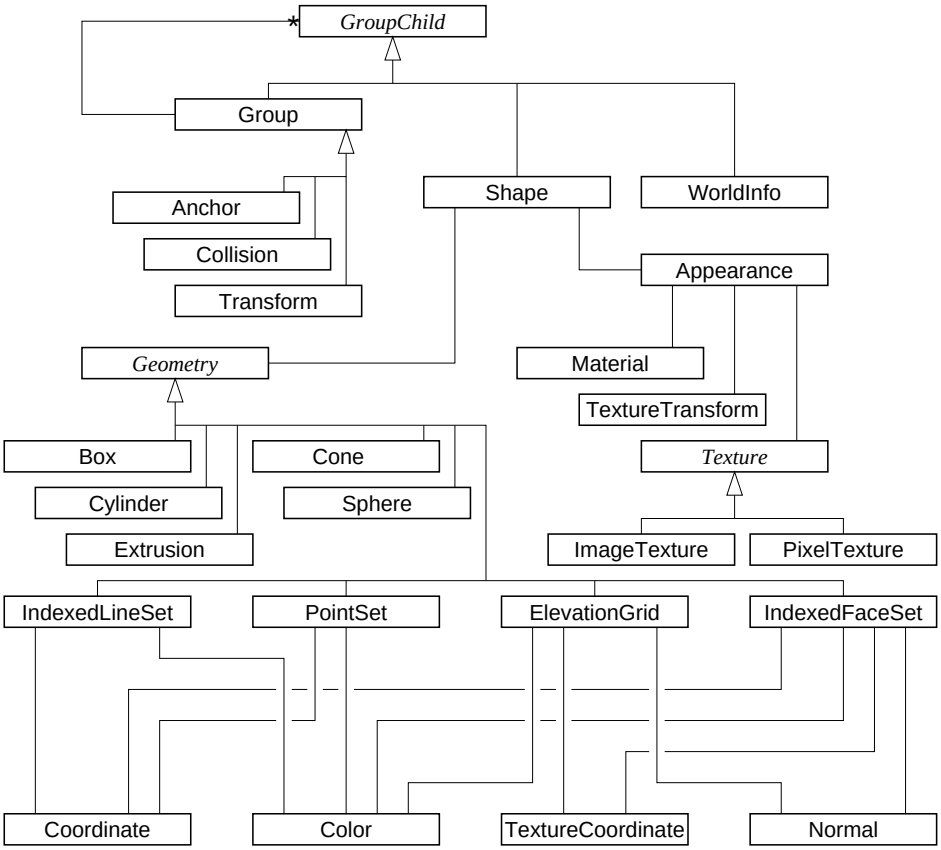
Les classes d'*ARéVi*

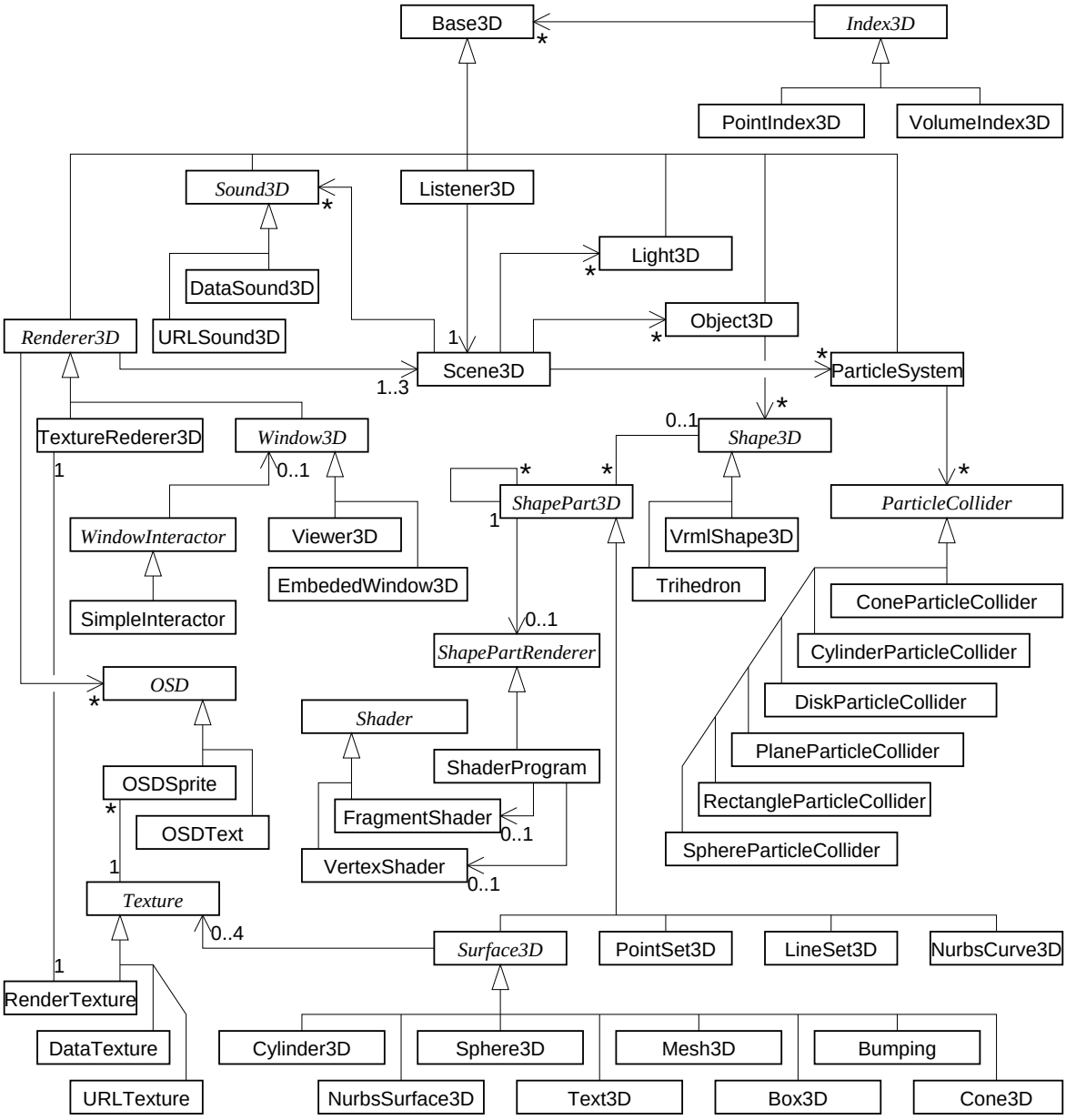
Nous donnons ici un aperçu des classe disponibles dans *ARéVi* (hors contributions).

```
ArSystem                  [initialization, system programming ... + main loop]

ArRef<>/ArConstRef<> [pointer substitutes (references) for garbage collection]
ArPtr<>/ArConstPtr<>                                          [weak references]

ArObject              [base class (GC, name, messages, serialisation, errors)]
 |
 |_ ArClass                [class reflection, name service, object management]
 |_ ArConstant                                            [constant reflection]
 |_ ArMethod                                  [method reflection and invocation]
 |_ ArType                                                  [type reflection]
 |_ ArValue                                    [value reflection and embedding]
 |
 |_ Scheduler                                             [activity sheduling]
 |   |_ RealTimeScheduler                          [conforming to user's clock]
 |   |_ VirtualTimeScheduler                      [conforming to time accuracy]
 |_ Activity                 [invoque objects methods conforming to scheduler]
 |
 |_ Message                             [support for asynchronous communication]
 |_ MessageService                                          [messages delivery]
 |   |_ NetworkMessageService              [messages delivery across network]
 |
 |_ AbstractIStream                                           [generic input]
 |   |_ FDIStream                                    [file descriptor input]
 |   |   |_ FileIStream                                        [file input]
 |   |   |_ URLStream                                    [file/http input]
 |   |_ MemoryIStream                                [memory block input]
 |_ AbstractOStream                                         [generic output]
 |   |_ FDOStream                                   [file descriptor output]
 |   |   |_ FileOStream                                       [file output]
 |   |_ MemoryOStream                                 [memory block output]
 |
 |_ MemoryBlock                                                [memory area]
 |   |_ RigidMemoryBlock                [not moveable/resizeable memory area]
 |   |   |_ MappedFile                                 [memory mapped file]
 |
 |_ IPSocket                       [high level sockets, blocking/not-blocking]
 |   |_ UDPTransmitter             [UDP send/receive/broadcast text and bytes]
 |   |_ TCPListener                   [TCP server waiting for TCP connections]
 |   |_ TCPConnection   [TCP send/receive text and bytes, with/without limits]
 |
 |_ RegularExpression                        [POSIX regular expression engine]
 |
 |_ Process                             [external commands with IO redirections]
 |
 |_ Thread                                                      [POSIX threads]
 |_ ThreadMutex                                            [POSIX thread mutex]
 |   |_ RecursiveThreadMutex               [recursive mutex (mutex+condition)]
 |_ ThreadCondition                                     [POSIX thread condition]
 |
 |_ InputDevice                   [generic input device with buttons and axes]
 |   |_ SpaceMouse                                          [6 DOF mouse usage]
 |   |_ Joystick                                               [joystick usage]
 |
 |_ JavaObject                    [peer to peer AReVi/Java object association]
 |_ JavaData                          [generic data to be exchanged with Java]
 |
 |_ TclCommand                                           [AReVi/Tcl association]
```

```
ArObject
 |
 |_ Base3D                          [3D geometry, conversions, attachements ...]
 |   |_ Object3D        [object with a 3D representation (billboard, LOD ...)]
 |   |_ Light3D                      [3D light (directionnal/positionnal/spot)]
 |   |_ Renderer3D                               [abstract display of 3D scenes]
 |   |   |_ TextureRenderer3D                                 [off-screen display]
 |   |   |_ Window3D                    [display in a window and manage events]
 |   |   |   |_ EmbeddedWindow3D               [embedded in an external window]
 |   |   |   |_ Viewer3D                                       [standalone window]
 |   |_ Sound3D                                  [located/oriented sound source]
 |   |   |_ URLSound3D                            [pre-existing sound ressource]
 |   |   |_ DataSound3D                              [generated sound ressource]
 |   |_ Listener3D             [location to listen to the sounds in the scene]
 |   |_ ParticleSystem                                       [particle system]
 |
 |_ WindowInteractor               [abstract interaction manager for Window3D]
 |   |_ SimpleInteractor                         [general purpose interactions]
 |
 |_ ParticleCollider                 [particle interactions (bounce, destroy)]
 |   |_ ConeParticleCollider                          [specific primitives ...]
 |   |_ CylinderParticleCollider
 |   |_ DiskParticleCollider
 |   |_ PlaneParticleCollider
 |   |_ RectangleParticleCollider
 |   |_ SphereParticleCollider
 |
 |_ Scene3D            [contains objects, particle systems, lights and sounds]
 |
 |_ Index3D [abstract dynamic structure for Base3D detection in constant time]
 |   |_ PointIndex3D           [ register points / detect points in a volume ]
 |   |_ VolumeIndex3D   [ register volumes / detect volumes covering a point ]
 |
 |_ Shape3D                                    [abstract 3D shapes for Object3D]
 |   |_ VrmlShape3D                  [shape conforming to a VRML2 description]
 |   |_ Trihedron                                          [utilitary trihedron]
 |
 |_ ShapePart3D                    [abstract tree structured part of a Shape3D]
 |   |_ Surface3D  [abstract surface (triangles, normals, material, textures)]
 |   |   |_ Mesh3D                                                [hand-made surface]
 |   |   |_ Box3D                                          [specific primitives ...]
 |   |   |_ Bumping
 |   |   |_ Cone3D
 |   |   |_ Cylinder3D
 |   |   |_ NurbsSurface3D
 |   |   |_ Sphere3D
 |   |   |_ Text3D
 |   |_ LineSet3D                                                      [polylines]
 |   |_ NurbsCurve3D                                                [nurbs polyline]
 |   |_ PointSet3D                                                         [points]
 |
 |_ Texture                                                        [abstract texture]
 |   |_ URLTexture                              [pre-existing texture ressource]
 |   |_ DataTexture                                [generated texture ressource]
 |   |_ RenderTexture                           [generated by TextureRenderer3D]
 |
 |_ ShapePartRenderer     [abstract alternative render method for ShapePart3D]
 |   |_ ShaderProgram                                    [shader based rendering]
 |_ Shader                                                 [abstract GLSL shader]
 |   |_ VertexShader                                       [pixel shader program]
 |   |_ FragmentShader                                  [fragment shader program]
 |
 |_ Material3D                                [describe a material for Surface3D]
 |_ Transform2D                               [describe a texture transformation]
 |_ Transform3D                                    [describe a 3D transformation]
 |_ BoundingBox3D                                  [describe a bounding-box]
 |_ Tesselator                    [extract triangles from a polygon (uses GLU)]
```

```
ArObject
 |
 |_ OSD                                          [abstract on-screen display]
 |   |_ OSDSprite                                [image above the 3D display]
 |   |_ OSDText                                   [text above the 3D display]
 |
 |_ VrmlInteractor          [allows to read/write VRML node fields (abstract)]
 |   |_ VrmlGenericInteractor               [allows navigation in a VRML tree]
 |   |_ VrmlBoxInteractor                      [specific node interactors ...]
 |   |_ VrmlColorInteractor
 |   |_ VrmlConeInteractor
 |   |_ VrmlCoordinateInteractor
 |   |_ VrmlCylinderInteractor
 |   |_ VrmlImageTextureInteractor
 |   |_ VrmlIndexedFaceSetInteractor
 |   |_ VrmlInlineInteractor
 |   |_ VrmlMaterialInteractor
 |   |_ VrmlNormalInteractor
 |   |_ VrmlShapeInteractor
 |   |_ VrmlSphereInteractor
 |   |_ VrmlTextureCoordinateInteractor
 |   |_ VrmlTextureTransformInteractor
 |   |_ VrmlTransformInteractor
 |   |_ VrmlWorldInfoInteractor
```




Base3D
Index3D
PointIndex3D
VolumeIndex3D
Sound3D
Listener3D
Light3D
DataSound3D
Renderer3D
URLSound3D
Object3D
Scene3D
ParticleSystem
TextureRederer3D
Window3D
Shape3D
ShapePart3D
ParticleCollider
WindowInteractor
Viewer3D
VrmlShape3D
EmbededWindow3D
Trihedron
SimpleInteractor
ConeParticleCollider
ShapePartRenderer
CylinderParticleCollider
OSD
DiskParticleCollider
Shader
PlaneParticleCollider
OSDSprite
ShaderProgram
RectangleParticleCollider
FragmentShader
OSDText
SphereParticleCollider
VertexShader
Texture
Surface3D
PointSet3D
LineSet3D
NurbsCurve3D
RenderTexture
DataTexture
Cylinder3D
Sphere3D
Mesh3D
Bumping
URLTexture
NurbsSurface3D
Text3D
Box3D
Cone3D

Index