EXOOD4 Tutorials
Video games development

Règles d'écriture des programmes

DIVERS

Table des matières

Note : Les règles présentées ont pour but d'être utilisées dans le cadre de la programmation en langage C ou C++.

L'écriture d'un programme (aussi complexe soit-il) nécessite quelques règles d'écriture. Ne serait-ce que pour pouvoir relire soi-même son code et surtout, dans le cadre d'un travail en équipe, pour permettre à d'autres programmeurs n'ayant pas écrit le code d'origine, de le relire, le modifier et le corriger facilement.

Reprendre plusieurs routines après plusieurs semaines ou plusieurs mois peut également être un travail fastidieux si le programmeur doit déchiffrer son propre code ! Alors autant prendre de bonnes habitudes dès le départ et éviter ainsi de perdre un temps précieux.

1 Les conventions d'écriture

1.1 "Notation Hongroise" (Hungarian notation)

Cette notation, inventée à l'origine par Charles Simonyi (chez Microsoft) a pour but de "normaliser" la programmation sur Windows. Elle permet de représenter le type d'une variable en y ajoutant dans le nom un préfixe plus ou moins significatif.

Exemples :

Notation	Description
i	integer
f	float
d	double
l	long
c	character
b	boolean
dw	double word
w	word
by ou byte	byte
sz	C-style null-terminated string
str	C++ string object
h	handle
v	vector
pt	point
rgb	RGB triplet
p	pointer to
r	reference to
u	unsigned
a ou ary	array of
m_	member variable
g_	global variable
s	static variable

On peut alors écrire les variables comme ceci :
fRadius, **sz**WindowName, **rgb**BackgroundColor, **m_sz**CharacterName, **g_l**MaxElements …

Bien que cette notation comporte plusieurs avantages, elle n'en reste pas moins inadaptée lorsque nous sommes confrontés au problème de portabilité ou encore lorsqu'il est nécessaire de changer le type d'une variable.

En supposant que l'on déclare une variable double word **dwRadius** et que l'on utilise cette valeur assez souvent dans le code, que se passerait-il si nous voulions utiliser un flottant plutôt qu'un double word ! Il faudrait modifier toutes les occurrences de la variable **dwRadius** en les remplaçant par la nouvelle variable **fRadius** ! Opération trop lourde, et parfois risquée.

On peut tout de même conserver certaines de ces notations, en particulier celles qui sont indépendantes de l'architecture de la machine (b, m_ , str, rgb etc…)

1.2 Nommer les identifiants

Le choix des identifiants de variable (ou de fonction) doit être le plus explicite possible. Il faut bien entendu éviter de tomber dans la facilité en nommant les variables a, b, c, i, j, compteur, nb etc… Il n'y a pas plus énervant que d'essayer de chercher le sens de telles variables (***compteur*** est un compteur de quoi ?? , ***nb*** représente quel nombre ??)

Nous savons qu'un identifiant de variable est composé de caractères alphanumériques, le premier devant obligatoirement être une lettre [A-Z][a-z] ou underscore _ , les caractères suivant pouvant alors être également un chiffre [0-9].

On retrouve des constructions différentes suivant le style de chacun :

- Certains préfèrent utiliser une majuscule pour le début de chaque mot significatif dans le nom d'une variable, les autres restant en minuscule (ex : *CompteurObjetsScene*)

- D'autres commencent par une minuscule et suivent ensuite la même règle (*compteurObjetScene*)

- Ou encore, utiliser les underscores pour séparer les mots (*Compteur_Objet_Scene, compteur_objet_scene …*)

- etc …

Dans l'écriture d'un moteur ou d'une librairie ou tout autre regroupement de modules et de fonctionnalités, il peut être utile de rajouter un préfixe commun. Ce préfixe sera le plus souvent utilisé pour les types et les noms de fonctions, mais on peut éventuellement le retrouver dans les identifiants de constantes ou tout autre valeur "importante" (globale). Cela permet également d'éviter les conflits de noms avec d'autres librairies.

Exemple : en supposant que le moteur se nomme Engine3D, on écrira :

Engine3D_Object, *Engine3D_Vector*, …
ou *E3D_Object*, *E3D_Vector*, …
ou encore *E3DObject*, *E3DVector*, …

Remarque : éviter les redondances dans les identifiants, cela rallonge leur écriture et surcharge la relecture.

Exemple :

TexturesManager->InitializeManager(…) ➔ TexturesManager->Initialize(…)

Nous savons à priori que l'objet TextureManager est un gestionnaire de textures, il est donc inutile de répéter une deuxième fois le mot Manager pour la fonction d'initialisaton.

2 L'écriture de programmes

En C++ un programme est composé de déclarations (variables, types, constantes, classes), de fonctions/méthodes, de directives de compilation etc…

Les commentaires ont une importance à ne pas négliger. Ils permettent avant tout de décrire le fonctionnement des algorithmes utilisés, mais aussi de présenter les différentes fonctionnalités d'un module, le rôle des variables, la description des types déclarés etc…

```
////////////////////////////////////////////////////////////////////////
// MyClass.cpp
//      Description    : Implémentation de la classe MyClass …
//      Last update    : 01/01/2003
//      Author         : Moi
////////////////////////////////////////////////////////////////////////

#include "MyClass.h"
…
////////////////////////////////////////////////////////////////////////
// Description de Methode1
// Rôle de Parametre1
// Rôle de Parametre2
////////////////////////////////////////////////////////////////////////
void MyClass::Methode1(int Parametre1, int Parametre2)
{
        int VariableLocale1;          // Description de VariableLocale1
        int VariableLocale2;          // Description de VariableLocale2
        …
}
…
```

Il existe également différentes écritures de blocs. Certains préfèrent utiliser les accolades comme ceci :

```
if (condition)
{
}
else
{
}
```

ou encore :

```
if (condition) {
}
else {
```

Pour une question de lisibilité la première solution paraît plus adaptée.

Dans le cas d'une seule instruction par condition il semble plus lisible de regrouper sur une seule ligne les tests+actions :

```
     if (condition1)  action1;
else if (condition2)  action2;
else if (condition3)  action3;
else                  actionParDefaut;
```

De même pour la sélection switch :

```
switch (condition)
{
        case 1 : action 1; break;
        case 2 : action 1; break;
        case 3 : action 1; break;
        ...
        case n : action n; break;
        default : actionParDefaut;
}
```

Remarque :
Attention à ne pas abuser de ces regroupements. Même s'ils semblent plus clairs à la lecture, ils ne sont pas adaptés lors de la mise au point d'un programme (debug). En effet, lorsque le programmeur souhaite tracer pas à pas son code, l'éxecution d'une ligne entraîne le traitement de plusieurs instructions (ce qui peut parfois être gênant).

2.1 Exemple

Voyons comme exemple une fonction écrite dans un premier temps de manière "illisible", et dans un deuxième temps de façon plus claire :

```
// 1ère version
int Rechercher (int v, int *t, int m)
{
        int a=0, b=m-1, c;
        while (a <= b) {
                c = (a+b)>>1;
                if (v > t[c]) a = c+1;
                else if (v < t[c]) b = c-1;
                else return c;
        }
        return –1;
}
```

Les habitués reconnaîtront surement l'algorithme de recherche dichotomique mais bon, est-ce vraiment clair ??

```
// 2ème version
int RechercheDichotomique (int _ValeurRecherche, int *_TabValeurs, int _MaxElements)
{
        int debut = 0;                  //  indice du début de l'intervalle de recherche
        int fin = _MaxElements - 1;     //  indice de fin de l'intervalle de recherche
        int milieu;                     //  indice du milieu de l'intervalle de recherche
```

```
        while (debut <= fin)
        {
                milieu = (debut + fin)>>1;          //  équivalent à milieu = (debut + fin)/2

                     if (_ValeurRecherche > _TabValeurs[milieu])        debut = milieu + 1;
                else if (_ValeurRecherche < _TabValeurs[milieu])        fin = milieu - 1;
                else                                                    return milieu;
        }

        return –1;
}
```

2.2 Choisir ses propres règles

Libre à chacun de choisir ses conventions d'écriture, du moment que l'on reste cohérent tout au long du programme.

Il existe des outils d'aide à l'édition de code (pour Visual C++) qui facilitent grandement la présentation des sources. Par exemple :

- Visual Assist (payant) : http://www.wholetomato.com/
  Comportant des outils de coloration des sources, se basant églament sur les déclarations de l'utilisateur (bien plus complet que la coloration de base de Visual C++). Génération de code à l'aide de templates (générer les en-têtes de programme, squelette des conditions, fonctions etc…). Complétion et correction automatique des fautes de frappes (reconnaissance des identifiants), etc …

- CodeTemplate (gratuit) : http://codeguru.earthweb.com/devstudio_macros/CodeTmplEx.shtml
  un AddIn pour Visual C++ qui permet de générer du code à l'aide de templates

Pour trouver son propre style de programmation, il peut être nécessaire de changer plusieurs fois jusqu'à trouver celui qui nous convient le mieux, celui dans lequel on se sent le plus à l'aise, celui qui se lit le mieux. Dans les différents articles de la rubrique Tutorials j'utiliserai les conventions suivantes :

- Les variables globales seront préfixées par la lettre **g**_ (ex: **g**_Variable)
- Les variables statiques par la lettre **s**_ (**s**_Variable)
- Les paramètres de fonctions par un underscore '_' (_Parametre)
- Les attributs d'une classe par **m**_ (**m**_Attribut)
- Les pointeurs par **p** (**p**Object)
- Les handles par la lettre **h** (**h**Window)
- Les gestionnaires seront suivis du mot **Manager** et seront des objets globaux "singleton" (**g**_Texture**Manager**) : voir les articles correspondant aux gestionnaires
- Généralement, tous les identifiants commenceront par une majuscule, sauf exception (VariableLocale, ExempleDeFonction, ExempleMethode …)
- + d'autres règles plus spécifiques qui seront présentées le moment venu