EXOOD4 Tutorials
Video games development

Création d'un profileur

MOTEUR DE JEU

Table des matières

1 Introduction

Le développement d'applications, d'outils ou encore de jeux vidéo nécessite très souvent une (ou plusieurs) phase(s) d'optimisation. Cela peut dans certains cas être primordial pour le résultat final, notamment dans le cadre d'un jeu vidéo ou d'une application exécutant beaucoup de calculs.

Il existe quelques outils d'aide à l'optimisation, comme le logiciel VTune ou tout simplement le Profiler de Visual C++, mais ces outils ne sont pas toujours adaptés aux besoins du développeur qui peut parfois avoir besoin d'autres informations ou une évaluation plus rapide de ses routines. En effet, l'utilisation de logiciels comme VTune ralentit l'exécution du programme car il doit analyser le code et évaluer chacune de ses instructions.

C'est pourquoi nous allons voir comment réaliser un mini-profileur, certes modeste, mais utile malgré tout. La précision du temps d'exécution estimé pour chaque fonction ne peut être comparée à celle des outils professionnels mais cela nous permettra de repérer et d'optimiser les parties critiques de nos programmes.

2 Construction du Profileur

2.1 Evaluer les besoins

Nous devons dans un premier temps décrire les résultats que nous souhaitons obtenir :

- de quelles informations avons-nous besoin
- comment seront représentées ces informations
- comment utiliser le profileur dans un programme

Nous allons donc définir une liste d'informations susceptible de nous intéresser : le temps d'exécution d'une partie de code, le nombre d'appels de la partie de code à évaluer et pour finir, un nom quelconque permettant d'identifier facilement la partie de code analysée.

Le profileur écrira dans un fichier de sortie toutes ces informations, en fin d'exécution du programme, sous la forme suivante :

```
[date] [time]> Session started

Profiler summary:

      ---------------------------------------------
      %      t(ms)  calls  Section
      ---------------------------------------------
      [%]    [time] [calls][Section Name 1]
      [%]    [time] [calls][Section Name 2]
      …
      [%]    [time] [calls][Section Name n]
      [%]    [time] -      other
      ---------------------------------------------
      Total  [time] ms


[date] [time]> Session closed
```

Remarque : chaque zone écrite en bleu sera remplacée par sa valeur correspondante.

Pour construire notre profileur nous allons écrire une classe **xod4ProfileManager** de type **Singleton**, qui elle-même utilisera une autre classe **xod4LogFile** lui permettant d'écrire toutes les informations dans un fichier de sortie. Nous aurons besoin des fonctions suivantes pour le Profileur :

- RAZ du profileur (réinitialisation) → ProfileReset
- Démarrer l'évaluation d'une portion de code → ProfileBegin
- Terminer l'évaluation d'une portion de code → ProfileEnd
- Ecrire les informations dans le fichier de sortie → ProfileReport

2.2 Implémentation de la classe ProfileManager

Nous allons tout d'abord déclarer une structure regroupant les informations de section :

```
typedef struct
{
	ulong	TimeSum;			// Total execution time of the section
	ulong	PreviousTime;		// Time of the section previously started
	ulong	CallsCount;			// Calls counter
	BOOL	bStarted;			// Set to 1 if the section is started
} xod4ProfileSectionInfo;
```

A l'aide d'un tableau de structures **xod4ProfileSectionInfo** nous pourrons évaluer plusieurs sections en même temps. En déclarant une série d'identifiants (numériques) il sera possible d'accéder directement à la section voulue. Nous choisirons donc des noms d'identifiant explicites.

Voici un exemple de déclaration des identifiants de section :

```
enum
{
// Section identifiers (put here the name of your sections)
	ProfileSection1 = 0,
	ProfileSection2,
	ProfileSection3,
	ProfileSection4,

// Keep this line with your declarations
	ProfileSectionCount
};
```

La dernière ligne de l'énumération doit toujours être présente lorsque vous déclarez des identifiants. La valeur **ProfileSectionCount** correspond au nombre d'identifiants déclarés et permet d'initialiser le Profileur.

Pour terminer, nous déclarerons un tableau de chaînes indiquant les noms "en clair" de nos différentes sections :

```
const char *g_ProfileSectionNames[] =
{
	"Section1",
	"Section2",
	"Section3",
	"Section4"
};
```

Vous pouvez voir maintenant le code qui accompagne cet article pour l'implémentation de la classe : *xod4ProfileManager.h* et *xod4ProfileManager.cpp*

Remarque : la classe Singleton fait partie des Design Patterns, il s'agit d'une technique très souvent utilisée en programmation orientée objet. Elle permet notamment d'avoir qu'une seule instance de classe possible dans un programme. On l'utilise principalement pour des gestionnaires (Managers) : gestionnaire de textures, gestionnaire de sons etc.

Nous ajouterons cette classe à la librairie "Exood4" (que l'on enrichira au cours des articles afin d'obtenir une librairie complète de développement).

*Remarque : vous pourrez constater que de nombreux identifiants ont pour préfixe "**xod4**" ceci étant fait pour personnaliser la librairie.*

3 Exemple d'utilisation

Nous prendrons comme exemple l'évaluation de 2 fonctions de tri (tri par insertion et le tri rapide "QSort"). A partir d'un même ensemble de valeurs nous comparerons les résultats obtenus.

Suivant les indications fournies dans le chapitre précédent nous allons déclarer les données nécessaires au profileur :

```
enum
{
// Sections identifiers

	SectionInsertionSort=0,
	SectionQSort,

// Keep this line with your declarations

	ProfileSectionCount
};

// Name of profile sections

const char *g_ProfileSectionNames[] =
{
	"InsertionSort",
	"QSort"
};
```

Vient ensuite la déclaration des fonctions de tri :

```
void InsertionSortAlgorithm(long _Array[], long _ArraySize);
void QSortAlgorithm(long _Array[], long _Low, long _High);
```

L'évaluation de ces fonctions s'effectuera de la manière suivante :

```
...
xod4ProfileBegin(SectionInsertionSort);
	InsertionSortAlgorithm(Array,size);
xod4ProfileEnd(SectionInsertionSort);
```

```
...
xod4ProfileBegin(SectionQSort);
        QSortAlgorithm(Array,0,size);
xod4ProfileEnd(SectionQSort);
...
```

Array étant le tableau de valeurs à trier. Il sera bien entendu le même pour les 2 fonctions de tri.

Nous obtenons alors les résultats suivant (dépend bien entendu de la machine sur laquelle sera exécuté le programme) :

Profile.log

```
03-01-2004 15:19:18> Session started


Profiler summary :

       ------------------------------------------------
       %      t(ms)  calls  Section
       ------------------------------------------------
       85     5735   10000  InsertionSort
       13     921    10000  QSort
       0      16     -      other
       ------------------------------------------------
       Total  6672 ms


03-01-2004 15:19:25> Session closed
```

Comme nous pouvons le constater la fonction QSort est bien plus rapide que le tri par insertion (ce qui semble logique). La section "*other*" n'est autre que le programme lui-même hormis les fonctions identifiées.

La classe **xod4ProfileManager** pourra donc nous servir dans le but d'optimiser certaines parties de code. Libre à vous de la modifier et de la personnaliser un peu pour répondre à vos attentes. Sur les mêmes principes il est très facile d'obtenir des outils complémentaires aidant à la mise au point de programmes. C'est ce que nous verrons très prochainement en rajoutant de nouveaux modules à notre librairie.

Retrouvez les sources de cet article à l'adresse suivante :

http://www.exood4.com/tutorials/articles/game_engine/Profiler.zip