PROJET ASTEROID BELT

Tuto Config Manager, fichier de config

Introduction

Ce tuto va vous servir à comprendre comment doit fonctionner votre Config manager et vous donne des tips pour l'implémenter.

Le config manager est censé être agnostique (ne rien connaître des détails de votre jeu).

Nous reviendrons également sur comment s'y prendre pour lire le fichier de config.

Etape 0 – Lancez le projet Sample.

« First make it work, then make it work good, then make it work fast. »

Le projet Sample est un simple projet vide qui contient une boucle d'inputs vide et un dessin qui affiche un cercle rouge avec la SFML.

Assurez-vous qu'il fonctionne. Il va nous servir de « bac à sable » pour tester les nouvelles features que nous souhaitons ajouter à notre jeu. En effet il peut être difficile d'ajouter de nouvelles choses au milieu de l'architecture complexe du jeu final.

Une pratique courante est donc de tester d'abord cette feature à fond, en isolation, dans un projet de test « quick and dirty » afin de vérifier que notre idée fonctionne, et corriger les bugs inhérents si nécessaire avant de tenter de l'intégrer au reste du projet. C'est aussi ce qu'on appelle un POC (pour Proof Of Concept, ou « preuve que ça marche »).

Le projet sample sera notre étape « First make it work ». Une bonne idée serait de garder une copie « vierge » du projet Sample pour pouvoir tester une nouvelle feature plus tard sans être gêné par le code de tests précédents !

Etape 1 – Stocker les valeurs de configuration

Pour le projet Asteroids, il faut aller lire un fichier de config pour initialiser le jeu.

Le format du fichier de config est très simple et suit toujours la même logique :

- Un mot-clé
- Une valeur par ligne correspondant aux paramètres associés à ce mot-clé
- Le mot-clé « end » pour signifier que ce setting est terminé.

Pour les valeurs, il n'existe que deux types possibles :

- string, ou :
- float
- Mais un mot-clé peut avoir un nombre variable de valeurs.

C'est-à-dire que cette entrée est valide (va stocker 2 floats sous le mot-clé « parallax_far_params »:

```
parallax_far_params
0.75 # scale
15 # speed
End
```

Mais aussi celle-ci (va lire 5 strings sous le mot-clé « music »:

```
music
assets/music/EVA_Realizations.mp3
assets/music/EVA_RearView.mp3
assets/music/EVA_Shitsuboshita.mp3
assets/music/EVA_ZetaReticuli.mp3
assets/music/NightRaptor_Crash-land.mp3
end
```

Ou encore celle-là (va lire 1 float sous le mot-clé « player_firing_rate »):

player_firing_rate

5

end

Pour le moment, concentrons-nous sur comment nous allons stocker ces informations. Nous irons lire le fichier après.

Afin de se rappeler de ce qu'on lit, nous allons devoir stocker les valeurs associées à chaque mot-clé.

Nous savons que :

- Les mots-clés sont toujours des std ::string
- **Nous pouvons avoir un nombre variable d'arguments à lire**

Pour cela, on aimerait utiliser une **std ::unordered_map** avec en clé, une std ::string qui correspond à notre mot-clé. La unordered_map étant un container associatif, il sera facile de récupérer les données associées à un mot-clé, juste en lui passant le mot-clé.

Plusieurs questions se posent :

- **Pourquoi choisir une std ::unordered_map plutôt qu'une std ::map** ? Quelles sont leurs différences ? Les avantages de la std ::unordered_map ? C'est typiquement le genre de truc que vous préciserez dans votre Document Technique.
- Quel est le type de valeur pris par la unordered map ? Rappelez-vous de nos contraintes : on doit pouvoir stocker un **un nombre variable d'arguments.** Connaissez-vous une structure de données template qui permet de stocker un nombre variable d'arguments ? Je peux vous donner un indice : vous l'avez recodé !

Lorsque vous avez fait votre choix de valeur de map, il suffit d'avoir deux unordered_map : une pour les strings, l'autre pour les floats.

Voici ce à quoi vous devriez arriver vers la fin :

```
class ConfigManager
{
public:

    ConfigManager() = default;

    ~ConfigManager() = default;


    bool    ReadConfigFile(const std::string & fileName);
private:

    std::unordered_map<std::string, ??? >   m_stringConfigValues;
    std::unordered_map<std::string, ??? >   m_floatConfigValues;


};
```

(Ignorez la fonction ReadConfigFile pour le moment.)

On veut maintenant tester nos « keyword mapper » en leur fournissant des keywords.

Aussi, on veut pouvoir couvrir un maximum de cas d'utilisations possibles :

- Ajouter les mots-clés un par un
- Ou bien en ajouter plusieurs d'un coup (pour un type donné : string ou float).

On s'entend qu'on a deux « Keyword Mappers » :

- Un pour les string
- L'autre pour les floats

Comment donc aller récupérer le bon keyword mapper en fonction du type qu'on nous passe en paramètre ? On peut pour ça utiliser les templates et une **explicit template specialization.**

En gros :

- Si on nous passe le type string, on utilisera le mapper pour les string
- Si on nous passe le type float, on utilisera le mapper pour les float

De plus, pour éviter d'avoir à répéter le long type std ::unordered … nous allons utiliser un **using template**.

Voici un bout de code qui devrait vous aider :

```
public:
    template <typename T>
    void    AddKeyword(const std::string& keyword)
    {
        KeywordMapper<T>& keywordsMapper = GetConfigValues<T>();
        keywordsMapper.insert({ keyword, {} });
    }

    template <typename T>
    void    AddKeywords(const std::vector<std::string>& keywords)
    {
        KeywordMapper<T>& keywordsMapper = GetConfigValues<T>();

        keywordsMapper.reserve(keywordsMapper.size() + keywords.size());

        for (const auto & kw : keywords)
        {
            keywordsMapper.insert({ kw, {} });
        }
    }

private:
    template<class V>
    using KeywordMapper = std::unordered_map<std::string, std::vector<V> >;

    template <typename T>
    KeywordMapper<T>&   GetConfigValues() = delete;

    template <>
    KeywordMapper<std::string>& GetConfigValues()
    {
        return m_stringConfigValues;
    }

    template <>
    KeywordMapper<float>&   GetConfigValues()
    {
        return m_floatConfigValues;
    }


    KeywordMapper<std::string>  m_stringConfigValues;
    KeywordMapper<float>        m_floatConfigValues;
```

Sentez-vous libre de modifier les prototypes. (pour ajouter des valeurs, par exemple ?)

Etape 2 – Aller chercher la ou les valeurs d'un mot-clé

Vous l'avez sûrement deviné de par les étapes précédentes, pour stocker les valeurs associées à un keyword nous utiliserons un vector.

Nous devons maintenant coder une interface publique qui permettra au game d'aller interroger la config pour récupérer les valeurs d'un mot-clé.

Dans une bonne interface, la simplicité d'utilisation compte presque autant que la qualité de l'implémentation derrière.

Ajouter à votre ConfigManager deux fonctions publiques :

```
template <typename T>
const T&GetValue(const std::string& keyword) const;

template <typename T>
const std::vector<T>&    GetAllValues(const std::string& keyword) const;
```

N'oubliez pas qu'un mot-clé peut prendre 1 à N valeurs. Pour des raisons de simplicité d'utilisation, on veut deux versions de la fonction GetValue :

- Une qui ne va chercher qu'une seule valeur. Dans les faits, on sait que certains params (comme player_firing_rate) n'auront qu'une valeur. Mais comme nous stockons des vectors, cela signifie juste que nous devrions avoir un vector de size 1. Donc lorsque l'utilisateur ne veut qu'une seule valeur, récupérez cette valeur pour lui, et envoyez-lui juste la valeur et non le vector.
- Une qui va chercher le vector de valeurs entier, car on sait qu'il peut potentiellement y avoir plusieurs valeurs. Là, l'utilisateur sera intéressé pour aller récupérer tout le vector (pour les musiques par exemple).
- La gestion d'erreur en cas de keyword inconnu est à votre discrétion… tant que vous savez **justifier** votre décision. (vous pouvez modifier le prototype de ces fonctions si nécessaires)
- Hint : vous allez devoir ajouter des versions const aux fonctions précédentes.

Etape 3 – Lire le fichier de config

Nous devrions maintenant avoir toute l'infrastructure permettant de stocker et de récupérer des valeurs.

À présent, comment faire pour aller lire ce fichier de config ?

Vous pouvez d'ores et déjà créer une classe ConfigManager qui a une méthode publique

```
bool    ReadConfigFile(const std::string & fileName);
```

Qui va retourner true si la lecture du fichier de config s'est bien passée ou false en cas d'erreur.

On considère qu'un mot-clé sans valeur derrière mais avec un end n'est pas une erreur (c'est juste bête…)

Par contre, un mot-clé sans end derrière, ça c'est une erreur.

- La gestion d'erreur est à votre discrétion… tant que vous savez **justifier** votre décision. (vous pouvez modifier le prototype de ces fonctions si nécessaires)

Ensuite, rappelez-vous de votre Tower Defense et ouvrez le fichier. Voici une boucle basique qui va lire le premier mot de chaque ligne sur un std ::ifstream ifi :

```
std::string line;

while (getline(ifi, line))
{
    std::istringstream is(line);

    std::string name;
    is >> name;


} ≤ 1ms elapsed
```

Continuons pas à pas. Pour le moment cette fonction ne fait pas grand-chose à part lire des mots. On doit faire l'algorithme suivant :

- Lire un mot
- Initializer un vector (ou plutôt deux : un de strings et un de floats ?)
- Tant que (le mot sur la ligne suivant n'est pas « end ») :
    - Je lis une valeur
    - Si c'est une string : J'ajoute la valeur à mon vector de string
    - Si c'est un float : J'ajoute la valeur à mon vector de float
- Une fois que j'ai lu « end » :
    - Je prends celui de mes deux vector qui n'est pas vide et j'ajoute ces valeurs, sous le mot-clé correspondant, dans le KeywordMapper qui va bien.

Une fois de plus, le code de base pour faire ça va ressembler à celui du tower defense dont je vous fournis une variation :

```
std::string line;

while (getline(ifi, line))
{
    // todo...

    std::istringstream keywordLineStream(line);

    std::string keyword;
    keywordLineStream >> keyword;

    std::cout << "I read " << keyword << '\n';
    std::string paramLine;

    // TODO...
    while (getline(ifi, paramLine))
    {
        std::istringstream paramLineStream(paramLine);
        std::string param;

        if (param == "end")
            break;

        paramLineStream >> param;
        std::cout << "I read " << param << '\n';
        float fparm = std::strtof(param.c_str(), nullptr);
        // TODO...
    }

    // TODO...
}
```

Vous devez ajouter :

- Les vectors
- La gestion des lignes vides et des commentaires, et des erreurs en général
- La lecture de la valeur.
- L'écriture du keyword et de ses valeurs associées dans le bon keyword mapper.

Attention : pour lire les valeurs, vous allez devoir trouver une technique pour différencier une valeur float d'une valeur string. À la base, tout est une string ! Alors comment savoir ?

Vous n'avez pas le droit à std :: stof car elle lance une exception si la string lue n'est pas un float. Et on n'aime pas les exceptions ! Alors vous pouvez vous servir de std :: strtof, mais celle-ci renvoie 0 quand la valeur nue n'est pas un float…

Alors comment différencier les cas où ce n'est pas un float, et les cas où c'est _vraiment_ un 0 ? Je vous laisse méditer là-dessus.

Enfin, normalement tout est bon : vous avez lu le fichier de config, et du coup, tout vos keywords et toutes vos valeurs sont bien stockés dans vos keyword mappers, et grâce à vos fonctions GetValue, le reste du code a désormais accès aux configs que vous avez lues.

Pour être parfait, vous pouvez faire de votre ConfigManager un « ConfigService » pour rester en accord avec le reste de la nomenclature du projet.