Forum: Moteur de jeu GameBlender et alternatives
Topic: hARMful engine
Subject: Re: hARMful engine
PostÃ© par: Bibi09
Contribution le : 21/1/2020 17:01:24

Python est un langage interprété. Contrairement à certains langages comme C, C++ ou Rust, il n'est pas compilé. Autrement dit, Python n'est pas directement traduit en langage assembleur (~ instructions en bits) que le processeur est capable d'exécuter directement. Pour pouvoir interpréter du Python et donc rendre son code exécutable par l'ordinateur, il faut un programme intermédiaire appelée interpréteur qui s'occupe de faire cette traduction "texte vers assembleur".

De tels programmes d'interprétation existent, écrits en différents langages pour Python. CPython est écrit en C, Jython en Java, etc.

Pour utiliser du Python depuis un programme en C ou C++, on a justement une interface en C existante qui fait cette traduction. Celle-ci permet soit d'embarquer du code Python sous la forme d'une chaîne de caractère, soit d'importer un module Python depuis un fichier.

Ici le lien de la doc, mais le code est indigeste bien qu'ils précisent que c'est à cause de nombreuses vérifications.
https://docs.python.org/2/extending/embedding.html

A l'inverse, on peut étendre le langage Python à partir de code C ou C++ (ou encore d'autres sans doute). On écrit le code qui fait ce qu'on veut en C/C++ et on crée ensuite des fichiers autour qui vont servir à importer et utiliser ces fonctions ou classes. La raison de faire ça peut être multiple : performances, réutilisation du même code en natif ou en Python, etc.
Dans le cas du BGE, on récupère par exemple des données de la scène 3D (position d'objets). Les gars se sont pas amusés à synchroniser 40 fois les mêmes données : ils ont dû faire un wrapper du code C++ existant dans Blender.

Pour mieux expliquer tout ça, parlons de trois phases permettant d'appeler du code natif depuis un script (l'exemple est bidon* et très simpifié, c'est pour l'idée) :
1. Code natif float myFunction(float a, float b, int c) { // .... return result ; } 2. Wrapper void wrapperMyFunction(PyFloat a, PyFloat b, PyIntc, PyFloat& result) { float resultFloat = myFunction(a.value(), b.value(), c.value()) ; result.setValue(resultFloat) ; } 3. Script f1 = 0.2 f2 = 3.6 i1 = -6 result = 0. wrapperMyFunction(f1, f2, i1, result)
* (j'ai inventé les PyFloat, etc)

1. Tu as du code natif, disons une fonction en C++ que tu utilises déjà dans ton application. Elle a ses propres paramètres pour son calcul et retourner son résultat. Maintenant, tu veux aussi pouvoir l'exposer en Python...

2. Problème : l'interpréteur Python il ne la connaît pas et ne sait même pas comment

l'appeler !

Il faut en effet prendre le problème à l'envers du cas habituel.
Dans le cas du BGE habituel, tu cherches la fonction "OrthoProjectionMatrix(plane, matSize, axis)".
Tu vois qu'attend ces 3 paramètres, donc tu donnes ces valeurs lors de ton appel de fonction.
La signature de la fonction guide l'appel de fonction.

Dans le cas de l'extension d'un langage de script, c'est l'inverse : on sait comment va être appelée la fonction et c'est donc à nous de créer la signature avec les paramètres qui seront donnés.
L'appel de fonction guide la signature de la fonction.

Or, notre belle fonction décrite en 1. n'a pas forcément cette signature (encore moins s'il faut un type de donnée spécifique au langage de script comme je l'ai fait ici).
On va alors appeler notre fonction en 1. depuis notre nouvelle fonction en cours de développement : c'est typiquement ce que fait un "wrapper".

Elle va par exemple "convertir" les PyFloat en float, etc. Les donner à notre fonction native et emballer le résultat en PyFloat tel que souhaité.

3. Avec tout un tas de choses en plus de ce que j'ai fait en exemple bidon, on peut enfin utiliser notre fonction en Python.