Forum: Moteur de jeu GameBlender et alternatives
Topic: hARMful engine
Subject: Re: hARMful engine
PostÃ© par: Bibi09
Contribution le : 9/10/2021 13:09:46

Bonjour à tous !

Ca faisait longtemps que je n'avais pas posté sur ce topic.

J'avais du arrêter d'apprendre le Rust pour des raisons personnelles impliquant fatigue et manque de concentration. Je n'avais pas pu aller bien loin sur l'architecture ECS dont je percevais très mal la mise en œuvre.

Ces toutes dernières semaines, j'ai repris ce projet d'architecture autour d'un prototype que j'ai développé en Python. Le langage est simple et parfait pour prototyper. J'avais prévu de mettre ça en place autour d'un projet de mini-jeu 2D via pygame... mais ça c'était sans compter une mauvaise surprise liée à Python. Je ne peux pas continuer à développer sur Python tout en profitant des bienfaits de cette architecture et de tout le travail accompli.

Avant d'expliquer les soucis que je rencontre, je vais faire un petit point sur ce qu'est cette fameuse architecture ECS.

Il y a des années de ça, des moteurs 3D comme OGRE ou Irrlicht sont apparus sur la scène du logiciel libre. Ils utilisent la programmation orientée objet et, pour chaque type d'objet dans une scène 3D, on a une classe donnée. Ces moteurs profitent donc de l'héritage. Un souci avec ce fonctionnement, c'est le manque de flexibilité que cela peut engendrer.

Une autre vague de moteur, probablement lancée par Unity, propose une approche différente. Les objets ou entités de la scène 3D sont génériques (GameObject par exemple) et, au lieu d'un héritage, sont faits de "composants" qui leur apportent des fonctionnements différents. On peut par exemple attribuer un composant Caméra pour créer une caméra dans la scène 3D. Pour créer un mesh dans la scène, on pourra donc apposer un composant Mesh et un composant Material sur un autre "GameObject". On gagne donc en souplesse d'utilisation des objets de la scène 3D, comme par exemple changer le Material en cours d'exécution ou lui ajouter un composant pour en faire un RigidBody. Ainsi, la scène 3D peut très facilement être modifiée tout au long de sa vie.
Cependant, si elle apporte une plus grande flexibilité, cette architecture pose des problèmes notamment en terme de performances.

Il y a maintenant quelques années, une nouvelle gamme d'API graphiques sont apparues et seront probablement le futur de la 3D à moyen-long terme.
Les API "historiques" comme OpenGL et DirectX (11 et inférieurs) s'utilisent généralement dans un seul thread. Or, nos processeurs modernes possèdent de plus en plus de cœurs leur permettant de paralléliser les tâches. Dans un jeu vidéo, on peut avoir de nombreux threads comme

l'un qui gère le son, l'autre le réseau et le thread dit principal qui va gérer l'envoi des données à la carte graphique pour faire le rendu des frames. Cependant, cela sous-exploite le processeur et la carte graphique.
Avec les nouvelles API comme Vulkan (générique), DirectX12 (Microsoft) et Metal (Apple), plusieurs threads peuvent envoyer des commandes à la carte graphique. Il y a un empilement des choses à faire côté GPU de sorte qu'on utilise au maximum le GPU qui n'attend plus ou moins que le CPU lui envoie des instructions. Du côté du CPU, cela signifie aussi qu'il est possible de créer plusieurs threads pour gérer le rendu.

Le souci avec une architecture comme celle proposée par Unity, c'est qu'elle est se révèle assez complexe à utiliser dans un cadre multithreadé. Les GameObjects sont organisés en arbre, l'arbre de scène. Or, il est difficile de prévoir le nombre de threads à utiliser pour parcourir cet arbre, sa profondeur, sa complexité. D'autant plus que cet arbre pourra grandir ou rétrécir, s'élargir ou se condenser au fil de l'exécution. Du coup, la stratégie doit être aussi évolutive pour donner suffisamment de travail à chaque thread sans en donner trop à l'un et pas aux autres.

Afin de faciliter grandement la mise en place du multithreading dans un jeu vidéo, il vaut donc mieux partir sur une mise à plat. Et c'est là que l'architecture ECS apparaît. ECS pour Entité-Composant-Système. On retrouve donc les notions d'Entités et de Composants comme pour Unity, mais on va voir que ce sont des éléments complètement différents. Bien qu'il s'agisse stricto sensu de programmation orientée objet, on parlera ici de programmation orientée données. Donc tout langage de programmation permettant la POO pourra servir à implémenter l'architecture ECS.
- Les Entités (ou GameObjects) ne sont plus des éléments d'un arbre. Ce ne sont plus que de simples ID (identifiants) uniques, autrement dit, une valeur entière. A chaque création d'entité, l'ID est incrémenté afin de toujours produire une nouvelle valeur différente de toutes les autres créées précédemment. On peut également prévoir la réutilisation d'un ID quand une Entité est détruite.
- Les Composants de Unity contenait à la fois les données et le traitement de ces données. Dans l'architecture ECS, les composants sont réduits à la simple agrégation de données. Aucune logique, aucune fonctionnalité ne leur est directement implémentée. Les Composants sont évidemment liés à une Entité mais pas comme sous Unity. Au lieu de mettre le Composant directement dans le GameObject, on va simplement faire un lien sous la forme par exemple d'un système de [clé:valeur], où la clé est l'Entité et la valeur le Composant. On peut créer une classe qui s'occupera d'établir et conserver ces liens.
- Les Systèmes sont, eux, les éléments qui vont traiter les données des composants. On a donc une séparation entre données et traitement. On peut voir les Composants comme des éléments d'une base de données et les Systèmes sont la partie traitement de ces données. D'où l'orientation données de cette programmation.

La structure de l'architecture est donc à plat, sans arborescence. On peut donc placer ces éléments dans des tableaux. Un impact intéressant de cela, en plus de permettre un multithreading facile (on découpe les tableaux en autant de bouts que de threads), est lié au fonctionnement des processeurs. Quand ils récupèrent des données adjacentes en mémoire comme des données d'un tableau, ils peuvent les traiter beaucoup plus rapidement grâce à la mise en cache. Des données dans un arbre peuvent se trouver à des emplacements mémoires trop éloignés, ce qui oblige le processeur à faire des accès mémoire plus nombreux. Cela engendre donc une latence bien plus importante et donc des pertes de performance.

On peut évidemment se poser la question de comment structurer nos objets pour avoir des liens de parenté entre eux. La réponse est simple et coule de source quand on l'énonce : le lien de parenté est lui-même un Composant ! De même que la transformation locale d'un objet, et de tout autre aspect indispensable pour un objet 3D.
A chaque type de Composant, on crée le Système qui lui correspond et qui est dédié au traitement de ce type de Composant.

Ce point exhaustif étant fait, je reviens à l'implémentation Python que j'ai faite. Attention, je ne prétends absolument pas avoir fait un truc parfait. D'ailleurs, si vous cherchez des exemples d'architecture ECS, je pense que vous aurez quelques difficultés à en trouver. Le pourquoi est simple, chaque projet n'a pas forcément les besoins et l'architecture ECS n'est pas un bloc de code imposé. En gros, chacun implémente l'architecture ECS à sa façon, suivant ses besoins. Il s'agit donc plus d'un guide que d'un outil qu'on utilise clé en main. Ma solution est donc pertinente dans le cas de mes projets mais elle pourra ne pas répondre entièrement aux besoins d'un autre développeur.

Voici le dépôt pour quiconque voudrait y jeter un oeil : https://github.com/dcarlus/PyECS

En plus des Entités, Composants et Systèmes dont j'ai parlé, j'ai ajouté quelques notions supplémentaires. Celle de Job qui est commune dans l'architecture ECS. Un Job va traiter en multithreading un ensemble de Composants, de préférence indépendants les uns des autres. On choisit le nombre de threads pour exécuter chaque Job.
Les Jobs s'exécutent successivement les uns après les autres au cours d'une même frame. On peut donc les hiérarchiser pour pré-traiter des données variées (position d'un perso, santé et force d'un perso, ...), les utiliser dans un autre Job (traitement de l'IA basée sur la position et les stats du perso) et enfin rendre graphiquement le résultat (animation d'attaque, rendu 3D).

Enfin, pour simplifier au maximum la gestion des données, une classe World. Elle centralise toutes les mécaniques de l'architecture ECS : création et suppression des Entités et des Composants qui leur sont liés, exécution du code, etc.

Au tout début du message, j'évoquais mes difficultés m'obligeant à abandonner Python pour poursuivre mes essais. Cela est dû à CPython, l'interpréteur par défaut de Python et à ma connaissance, le plus complet vis-à-vis des paquets de bibliothèques disponibles. CPython gère le multithreading mais hélas, il ne fait tourner qu'un thread à la fois (voir le Global Interpreter Lock pour plus de détails). Si bien que mon application de test, qu'elle utilise 1 ou 16 threads, tourne tout aussi lentement avec de nombreux personnages.
https://en.wikipedia.org/wiki/Global_interpreter_lock

Or, l'objectif de ce test est non seulement l'implémentation fonctionnelle de l'architecture ECS (réussite avec Python) mais aussi mesurer les performances grâce au multithreading (échec avec Python). Il existe d'autres interpréteurs que CPython, ceux-ci offrant un multithreading plus performant. Cependant, j'ai essayé d'installer pygame avec PyPy3 sans réussir. J'ai aussi voulu compiler pygame avec PyPy3, mais là aussi ça a été un échec cuisant (sous Windows). Il me reste l'éventuelle solution de tester cela sous Linux, lequel pourrait me permettre d'y parvenir.
Bien que cette expérience me frustre à cause de ce deuxième point non honoré (pour l'instant), je suis malgré tout très satisfait d'avoir réussi à implémenter une architecture

ECS qui fonctionne et qui soit, je pense, assez simple d'utilisation. Il faut dire que c'était quelque chose de très abstrait pour moi et j'avais du mal à savoir comment l'appréhender.

Le projet ne s'arrête pas là. Je compte reprendre mon code Python pour le porter dans un autre langage comme C++ ou Rust que j'aimerais continuer d'apprendre. Je pourrai alors tester le multithreading en conditions réelles et avec des performances encore meilleures ! Je verrai pour faire peut-être un mini-jeu 2D à partir de là.