POLYTECHNIQUE
MONTRÉAL

LE GÉNIE
EN PREMIÈRE CLASSE

LOG4715

Guide de programmation

Version 22.3

Responsable : Olivier Gendreau

Chargé de laboratoire : Gabriel Paquette

Auteur initial : Jean-Marc Nielly

Chargement du cadriciel

1. Copiez le contenu de l'archive du cadriciel vers un dossier (évitez le disque réseau des ordinateurs de Poly), puis décompressez-le.
2. Si vous utilisez Unity Hub :
    - Cliquez sur l'onglet « Projects » situé à gauche. Cliquez ensuite sur « Open ». Naviguez jusqu'au cadriciel décompressé, cliquez sur le dossier du projet, puis cliquez sur le bouton « Sélectionner un dossier ». Cliquez sur le projet que vous venez de créer.
3. Si vous n'utilisez pas Unity Hub (comme au L-4818) :
    - Une fenêtre « Launch Unity » devrait ouvrir. Cliquez sur « Open Project ». Naviguez jusqu'au cadriciel décompressé, cliquez sur le dossier du projet, puis cliquez sur le bouton « Sélectionner un dossier ».
4. L'importation commencera et prendra quelques minutes. **Notez que le cadriciel fonctionne sous la version Unity 2021.3.4f1**

Références utiles

https://learn.unity.com/tutorials
https://docs.unity3d.com/Manual/

Notions essentielles de Unity

1. Entité-composante
    - Les entités visibles dans Unity sont toutes de la même classe dont il est impossible de dériver. Cette classe GameObject ne contient que quelques fonctionnalités rudimentaires :
        - Un système de messagerie.
        - Le *transform* qui permet de connaître sa position, sa rotation et son échelle.
        - Un patron composite permettant de structurer les GameObject de l'arbre de rendu.
        - Un nom.
    - GameObject est aussi capable d'accueillir des composantes qui ajoutent à ses fonctionnalités. Inclus dans Unity sont entre autres:
        - Le modèle 3D et le rendu visuel.
        - Les formes de collision et les comportements de physique de corps rigides.
        - Les fonctionnalités audio.
    - Nous pouvons aussi ajouter des composantes en dérivant de la classe MonoBehaviour. Les composantes MonoBehaviour attachées à un GameObject reçoivent des appels de fonctions aux moments opportuns:
        - Update: Une fois par trame.
        - OnCollisionEnter / OnTriggerEnter
        - Start: À l'apparition de l'objet en jeu.
        - OnEnable: À la (ré)activation de l'objet.
        - OnDisable
        - Liste complète: http://docs.unity3d.com/ScriptReference/MonoBehaviour.html

- Les MonoBehaviours peuvent communiquer entre eux de plusieurs façons.
  - GetComponent<type> trouve une composante d'un type donné sur un GameObject. On peut l'appeler sur le GameObject lui-même ou sur une de ses composantes.
  - GetComponentInChildren cherche la composante dans les enfants hiérarchiques du GameObject.
  - GameObject.Find cherche un GameObject par son nom (attention : cette fonction est plutôt lente. On peut l'utiliser sans problème lors de la création d'objets de la scène, mais elle est à proscrire dans la boucle de jeu principale).
  - SendMessage permet d'appeler une fonction par son nom sur un GameObject, peu importe sur quelle composante elle se trouve.
  - Physics/Physics2D permet de faire des Raycasts et des Spherecasts pour trouver les GameObjects ayant des Colliders dans une zone.
  - Bien sûr, il y a un coût de performance à tous ces appels. Rien ne vaut une référence directe si un lien est permanent, et rien n'empêche d'enregistrer les GameObject dans un gestionnaire pour créer un autre système de recherche.

2. Exceptions
   - Unity encapsule chaque appel à une composante dans un bloc *try-catch*. Les erreurs se retrouvent affichées sur la console.
   - L'erreur la plus fréquente est le NullReferenceException. Elle arrive quand on essaie d'appeler une méthode sur une référence vide (l'équivalent d'un pointeur à 0). En C#, il est fréquent de signaler l'absence d'un objet en retournant *null* dans un accesseur. Il est donc important de vérifier si une référence est égale à *null* lorsqu'applicable.
   - Dans le haut de la section console, vous trouverez l'option *Error Pause*. Cette option met votre jeu en pause lorsqu'une erreur se produit. Cette fonction est excessivement utile pour déboguer l'exécution de votre jeu.

Optimisation

1. Le *garbage collector*
   - Tout comme Java, C# utilise un *garbage collector*. Vous n'avez donc pas à faire la gestion de la mémoire vous-même. Il reste toutefois souhaitable d'éviter les allocations inutiles sur le *heap* à cause de leur lenteur et du travail de désallocation qu'elles représentent. Le *garbage collector* effectue son travail à des moments imprévisibles, ce qui peut mener à des retentissements si on le surcharge.
   - En jeu vidéo, le patron de conception Object Pool peut aider à éviter les allocations inutiles. Il s'agit de donner un état inactif à une entité lorsqu'elle ne sert pas et de le garder dans une réserve (*pool*) d'où l'on pourra la reprendre et la réinitialiser lorsqu'on aura besoin d'une entité de son type. En jeu vidéo, c'est souvent utilisé pour les ennemis (qui doivent mourir et être remplacés), les particules et les projectiles. (Source: http://gameprogrammingpatterns.com/object-pool.html)
   - Le *garbage collector* fonctionne par arbre, donc il n'est pas nécessaire de se soucier de références cycliques.
2. Valeur vs référence
   - En C#, les types sont soit par valeur, soit par référence. C'est implicite, selon le type.
     - Valeur:
       - bool
       - byte
       - char
       - decimal
       - double

        - enum
        - float
        - int
        - long
        - sbyte
        - short
        - **struct**
        - uint
        - ulong
        - ushort
    - Référence:
        - **class**
        - interface
        - delegate
        - object
        - string
  - Les références fonctionnent essentiellement comme des pointeurs intelligents en C++. Les types par référence sont toujours alloués sur la *heap* et laissent un pointeur là où ils sont déclarés. Une copie d'une référence pointe vers le même objet d'origine et donc modifie le même objet lors de modifications. Les types par valeur sont copiés par valeur, et donc une modification n'affecte pas la copie d'origine.
  - Les types par valeur sont alloués là où ils sont déclarés. S'ils appartiennent à la portée (*scope*) d'une fonction, ils sont alloués sur la pile (*stack*). S'ils font partie d'un objet (class ou struct), ils sont alloués directement dans l'objet. Dans ces deux cas, ils ne représentent presque pas de travail d'allocation et de désallocation. Il est donc important de se rappeler qu'une classe est un type par référence et qu'une struct est un type par valeur.
    (Source: http://www.c-sharpcorner.com/UploadFile/rmcochran/csharp_memory01122006130034PM/csharp_memory.aspx?ArticleID=9adb0e3c-b3f6-40b5-98b5-413b6d348b91)
2. Class vs struct
  - Considérez d'utiliser un struct pour un petit objet (moins de 16 octets) à la vie généralement courte, similaire à un *int* ou un *float*. L'utilisation d'un type par valeur peut éviter des efforts d'allocation et de désallocation.
  - Un objet boîte (*box*) doit être alloué et désalloué sur le *heap* lorsqu'on veut traiter un type par valeur comme un type par référence.
  - Comme pour tout en optimisation, il faut tester!
    (Source: http://msdn.microsoft.com/en-us/library/ms229017.aspx)
3. Tableaux
  - Un tableau de types par valeur est généralement plus rapide lorsqu'on itère qu'un tableau de types par référence. C'est parce que ce sont les références qui sont dans le tableau, les objets eux-mêmes sont éparpillés en mémoire, ce qui mène à des *cache miss*. Les types par valeur peuvent, eux, être parfaitement adjacents en mémoire dans un tableau.
  - Le tableau multidimensionnel voxels[3,4,5] a des vérifications de sécurité plus nombreuses lorsqu'on l'utilise que le tableau unidimensionnel équivalent voxels[*length**3 + *height**4 + 5]. On peut donc économiser en performance en aplanissant les tableaux. Le tableau voxels[3][4][5] est fragmenté en mémoire et donc habituellement plus lent.