Le Paladin de l'échec

présente

My Tiny Warcraft

Kenan "Le Bleu" Lejosne

Avant propos

Ce tp est écrit intégralement à fins pédagogiques, faites en ce que vous voulez. Pour toutes questions, référez vous au mail de contact.

CONTACT
kenan.lejosne@epita.fr

Contents

1 Introduction

1.1 Outil

Pour ce TP, je considérerai que vous avez une version de unity supérieure ou égale à Unity 2017.2.0f3. Pour l'éditeur de texte, faites comme bon vous semble, mais je ne peux que vous conseiller d'en choisir qui dispose d'autocomplétion sur Unity.

Ne vous sentez pas obligés de faire du version control, mais si vous le faites, utilisez git.

Pour finir, ceci est un tp d'introduction, cela ne veut en aucun cas dire que tout ce qui est dit ici doit être vu comme du texte sacré de référence. Après tout le jeu vidéo est un domaine large et plusieurs solutions existent pour résoudre un problème, si vous êtes à l'aise avec Unity, déjà je ne sais pas pourquoi vous lisez ces lignes, mais sentez vous libres de dériver du tp pour implémenter vos propres solutions.

1.2 Description de l'interface de Unity

L'interface de Unity est relativement simple et épurée, contrairement à son cousin Unreal Engine 4 qui vous en met plein l'écran, Unity est relativement simple. Il sera ici question de l'interface dans son layout par Défaut, car celui ci est modulable. Si vous n'êtes pas certains d'avoir le Layout par défaut, allez dans le menu en haut Window/Layout/Default. La description va se faire de haut en bas de droite à gauche.

1.2.1 Outil de base

En haut à gauche vous pouvez voir un groupe de 5 boutons :

- **La main**: permet de click n drag sur le scène pour déplacer la caméra, depuis n'importe quel outil on peut simplement faire click milieu et drag pour avoir le même effet.
- **La croix fléchée**: permet d'appliquer une translation sur un objet dans le niveau. En clickant sur un axe ou un plan on peut limiter la translation au plan ou axe choisi.
- **L'Ouroboros**: (ouais je balance des trucs cools) pareil que la translation mais pour la rotation.
- **Le petit carré avec les flèches vers l'extérieur**: (j'avais pas mieux): pareil que pour les deux précédents mais pour l'homothétie, aussi appelé mise à échelle. Ou "Grossissement" par les paysans.
- **Le rectangle autour du rond**: sélection par rectangle. Sélectionne par rectangle de sélection tout les objets de la scène.

Ces 5 boutons sont mappé par défaut sur QWERT (je n'ai franchement pas tester si c'est mappé sur AZERT pour les claviers FR). Vous remarquerez qu'il y a aussi deux autres boutons sur le coté droit de ces 5 premiers. Le premier sert à indiquer le centre utilisé: pivot ou centre réel. Le pivot étant un autre point utilisé pour la rotation mais ne sera pas plus approfondi ici, je vous invite à ne pas y toucher, bien qu'il ne devrait rien faire car le pivot est par défaut au même endroit que le centre. Le second est le référentiel utilisé.

- **Local**: utilise le référentiel local, les axes dépendent de votre rotation.

- **Global**: utilise les axes globaux, indépendamment de votre rotation.

Si vous n'avez pas compris ce passage, retournez taffer votre physique, askip Frenet ça tombe pas.

Encore plus à droite vous pouvez voir trois boutons qui ressemblent à une vieille chaine hifi. Le bouton play lance votre jeu, pause bloque l'exécution du code (relançable avec play) et le next permet de juste aller à la frame suivante (sert uniquement en pause).

1.2.2 Hierarchy

En dessous des boutons, vous devriez voir un onglet nommé Hierarchy. Dedans vous trouverez quelque éléments comme le nom de votre scène à coté du logo Unity (probablement un nom par défaut si vous n'avez pas encore sauvegardé). En dessous sont listés tout les éléments de votre scène. les contrôles de sélection avec CTRL et SHIFT sont les mêmes que ceux Windows.

Petit point sur pivot/center. Si vous sélectionnez plusieurs éléments avec SHIFT, le référentiel LOCAL est celui du premier sélectionné et le point de pivot est le centre du premier sélectionné. Center va considérer le centre de votre d'ensemble d'objet. Bien que cela soit possible, je vous déconseille d'utiliser le pivot sur des groupes car cela peut faire des choses étrange sur rotation et surtout mise à niveau. Je vous invite à essayer si vous voulez en savoir plus.

La Hierarchy permet principalement de voir vos objets, de les grouper ou de rapidement accéder à un élément. Quelque tips utile: L'ordre dans la liste ne change rien. Double clicker sur un un objet dans la liste met la camera de la scène proche de ce premier. Si vous faites un click and drag d'un objet sur un autre, le premier sera enfant du second. Ce qui veut dire que toute transformation (translation, rotation, mise à échelle) sur le parent se

fera sur les enfants. Encore une fois, faites attention à la mise à échelle et la rotation, les résultat peuventdes fois être inattendus.

1.2.3 Game scene

A droite de la hierarchy, il devrait y avoir une GROSSE fenêtre qui vous fait de l'œil depuis le début. Le scène est la vue "éditeur" de votre scène. Vous pouvez y sélectionner vos objets, les déplacer, voir les colliders et autre. Si vous regardez en haut à gauche vous pouvez voir des onglets. Et là, vous devriez voir "Game". Oui, ce que vous cherchez est juste là: la vue du jeu! Alors oui c'est vide, mais en même temps ce document ne servirait à rien si tout était déjà là. Plus sérieusement, vous pouvez voir la vue de votre caméra principale sur cette onglet. quand vous lancez votre jeu, vous voyez par défaut cette onglet. Si celui-ci n'est pas visible, par exemple par défaut il est caché par la scène, il se mettra en premier plan. Il faut noter qu'il est toujours possible de voir votre scène pendant que le jeu tourne, cela peut être pratique pour par exemple faire des mouvements extérieur à votre logique de jeu (traverser des murs par exemples) ou mettre votre niveau dans un état particulier. Les modification en jeu sont perdus lors de l'arrêt du jeu. Pour éviter de perdre des modifications car vous avez oublié que votre jeu tournait (ça peut parraitre débile, mais des fois on ne s'en rend pas compte) je vous conseille d'aller dans le menu en haut Edit/Preferences et dans l'onglet Colors/General choisir une couleur flashy pour Playmode Tint. Ainsi lorsque votre éditeur sera en train de tourner votre jeu, il sera tout flashy moche, vous permettant de vous rappeler visuellement que vous êtes en jeu.

1.2.4 L'inspector

L'inspector est le meilleur gadget de l'interface, se situant par défaut à droite. Pour illustrer simplement, je vous invite à clicker sur la caméra, soit sur la scène soit sur la hierarchy. Vous devriez voir quelque chose apparaître dans l'inspector. En effet l'inspecteur sert à... inspecter. Il permet entre autre d'accéder aux différents components de votre objet. Les objets peuvent avoir plein de components, physique, son, texte, scripts etc... Mais tous ont au moins un Transform. Ce component est basiquement la représentation spatiale de l'objet. Il stock la position, rotation et échelle. La Caméra ne faisant pas exception à la règles vous devriez voir ces 9 valeurs. Vous pouvez les éditer et voir les changements. Seul la mise à échelle ne fait rien car une camera n'est pas "grossissable". Mais regardons plutôt le component Camera. Celui-ci à une icone, une petite flèche permettant de réduire et un carré coché. Ce carré est ce qui permet de dire si le component est actif ou non. Un component inactif agit comme si il n'existait pas. Vous pouvez voir plusieurs paramètres, nous reviendrons plus tard sur comment marche ces menus et comment en faire pour vos scripts pour éditer rapidement vos jeux. En bas un gros bouton vous permet d'ajouter un component en cherchant dans la liste des components de base en plus de ceux qui vous avez créer.

1.2.5 Project et Console

Finalement, en bas à gauche vous pouvez voir deux onglets, projet et console. L'onglet projet est un simple explorateur de fichier dans le dossier Assets de votre projet. Vous pouvez en ajouter d'autre mais cela finalement rien de plus qu'une copie de l'explorateur windows. La console est votre sortie standard. Elle vous permet de voir des warnings, des erreurs et des logs. il faut noter que le dernier message de la console est affiché en bas de la fenêtre sur la

barre grise.

1.2.6 Click droit

Si vous faites un click droit dans la hierarchy ou l'onglet projet vous pouvez voir qu'un menu de creation s'ouvre. Dans la hierarchy vous pouvez créer des objets simples pour votre scène et dans le projet des assets simples. Si vous faites un drag and drop un asset depuis le projet dans la scène vous créez une copie dans la scène. Si, à l'inverse, vous faites un drag and drop d'un objet depuis la scène dans votre projet vous créez une copie de votre objet en tant qu'asset. Mais qu'est-ce qu'un asset? Et bien c'est tout objets qui peut être utilisable directement dans votre jeu. Une porte, un mecha, une lumière. Le plus souvent les Assets font références aux assets graphiques qui sont, surprise, les objets graphiques de votre jeu. Mais théoriquement un script c'est aussi un asset.

1.3 La programmation sous unity

Maintenant que vous avez pris conscience du minimum qu'il faut savoir sur Unity et que vous avez un document écrit en plus de la conférence, on va pouvoir faire quelque rappels pour ceux qui dormaient sur globalement comment marche le code sur Unity.

1.3.1 Les GameObject

Si on omet le réseau (qui utilisera des NetworkBehaviour pour le système built-in), la plupart de vos script ayant pour vocation d'intéragir directement avec votre jeu hériteront de GameObject. Mais qu'est-ce donc qu'un GameObject?

Base class for all entities in Unity scenes.

Voilà. C'est la définition de la documentation de Unity. En réalité un GameObject est vraiment quelque chose de simple, c'est quelque chose que le moteur va gérer et qui sera influé par ce dernier. Pour donner quelque exemples: Un script qui permet de se déplacer doit être un GameObject car il doit interagir avec le moteur (pour déplacer quelque chose) et le moteur va aussi interagir en appelant des fonctions à chaque frame par exemple. En revanche, un script qui permet de récupérer le monstre le plus proche du joueur n'est pas nécessairement un gameObject. Certes il doit avoir connaissance des monstres, mais à partir de ces monstres, il n'a besoin en aucun cas d'interagir avec le moteur et peut donc être implémenté sans hériter de GameObject.
Il fut beaucoup question d'interaction avec le moteur, mais qu'est-ce qu'une classe héritant de GameObject peut faire?

- S'attacher à un objet et accéder à l'objet portant le script. En effet, le but de vos scripts est d'être attaché à un objet, par exemple un script de tir au héros. Non seulement être un GameObject vous permet de vous attacher, mais en plus de récupérer le GameObject de l'objet qui vous porte et faire des modifications.

- Avoir une existence dans la boucle de jeu. Un jeu vidéo c'est au centre une grosse boucle qui met à jour tout ses objets et essaie de tourner un maximum de fois possible (créant ainsi les fameuses frames par

seconde). Un objet C# n'est pas lié à cette boucle. Par contre un GameObject l'est. Non seulement il sera instancié par le moteur, mais en plus ce dernier appellera différentes fonctions à différents moments (début du jeu, à chaque frame, lors d'une collision etc...)

- Plus globalement, cela permet d'exister à l'échelle du moteur et d'en utiliser ses caractéristiques.

Vous aurez donc compris que vous allez beaucoup interagir avec ces fameux GameObject. En fait, tout votre gameplay sera dans des GameObject, sauf quelque classes de gestions. Et du coup, parce que c'est logique, pour être un GameObject, votre classe doit bien évidement hériter de Monobehaviour.

2 Le sujet

Il est temps de s'y mettre. On va faire un super jeu. Un RTS à l'ancienne, le genre de jeu qui fait rage en Corée. On va faire un mini-warcraft 3. Je dis mini parce que j'ai pas le temps pour vous faire un sujet qui vous explique comment en faire un vrai et encore moins le temps de faire les modèles 3d. Alors on va faire à l'ancienne. L'objectif est donc de reproduire les mécaniques de base d'un jeu de stratégie en temps réel avec des héros. Il y aura donc:

- Des unités simples
- Une Camera RTS
- Des héros qui peuvent gagner de l'expérience
- Des batiments destructibles
- Une ressource

3 Les Unités

Le centre de tout jeu de stratégie c'est bien évidement des unités, c'est ce que vous pouvez controler directement et c'est aussi ce qui vous permet de gagner vos parties en anéantissant vos adversaires. Ca serait donc pratique d'en avoir.
Le centre d'une unité c'est trois fonctionnalités:

- De la vie
- Le déplacement

- Le combat

En effet, sans l'une de ces trois composantes, il parait assez clair que votre unité aura du mal à servir à quelque chose sans être abusivement puissante. Une unité qui n'a pas d'interaction de combat ou de barre de vie étant invincible. Une unité sans déplacement peut exister, toutefois cela fait souvent plus référence à des bâtiments, chose que nous verrons plus tard.

Mais avant de se lancer dans l'implémentation tête baissée, réfléchissons rapidement pour éviter de copier du code ci et là. Dans un jeu, il y aura plusieurs entité, certaines sont factorisables, d'autres non. Dans jeu de stratégie on peut déjà faire la distinction entre deux grands groupes d'entité : ceux qui ont de la vie ~~et ceux qui creusent~~ et ceux qui n'en ont pas. Dans les choses qui n'ont pas de vie, on a les ressources (arbre et mine d'or sur warcraft) et les magasins (toujours sur warcraft). Il est donc relativement clair que l'on peut regrouper déjà toutes les unités dans une classe qui contiendra les méthodes de dégâts et de mort. Mais il ne faut pas oublier ceux qui n'ont pas de vie, pour cela il faut créer quelque chose qui regroupe les deux: une classe Entity. Ensuite, parmi les entités qui ont de la vie, on va pouvoir encore diviser en deux grands types: les bâtiments et les unités. En effet, les entités ayant de la vie ne se déplacent pas forcément, seules les unités peuvent se déplacer. Toutefois une unité EST UNE entité ayant de la vie n'est-ce pas? Si vous ne voyez pas trop comment faire cette relation, c'est parce qu'il faut faire de l'héritage, une notion que vous ne maîtrisez pas encore bien (et c'est pourquoi je vous file un coup de main). Vous croyiez que c'était bon? NON, en prévision de de l'implémentation des héros il va falloir créer encore une distinction entre les personnages qui peuvent gagner de l'expérience et ceux qui ne peuvent pas.

3.1 Entity

Pour permettre d'avoir une abstraction, nous allons créer une classe Entity. Cette classe sert simplement à réunir les entités, (qu'elles aient de la vie ou non). Pour cela, cette classe contiendra uniquement un ensemble de méthode que nous implémenterons plus tard pour l'interface. En attendant, créez simplement une classe Entity. Elle hérite bien évidement de Monobehaviour.

```
class Entity : MonoBehaviour
{
}
```

3.2 Life_Entity

Pour pouvoir faire des entités ayant de la vie, nous allons regrouper tout ce qui concerne la gestion de la vie dans cette classe. Cette classe contiendra donc un ensemble de chose nécessaire pour gérer la vie:

- Une variable protected de vie maximale
- Une variable protected de vie actuelle
- Une méthode publique pour faire des dommages (ne retourne rien)
- Une méthode protected qui gère la mort (ne retourne rien)

```
class Life_entity : Entity
{
    [SerializeField]
    protected int hp_max_;
    protected int hp_current_;
```

```
    protected void Death(){}

    public void apply_dommage(int _dmg){}

}
```

Pour ces méthodes assez simples, vous être libre d'implémenter comme vous le souhaitez. Vous pouvez aussi ajouter des choses, comme un système de régénération de vie par exemple.

Protected. Pour permettre aux classes filles de modifier le comportement de ces méthodes ou variables, plutot que de les mettre en private, mettez les en protected, cela permet d'avoir la même chose que private, sauf pour les classes filles.

Serialize Field. Beaucoup de personnes pensent que pour permettre à une variable d'être éditable directement depuis l'éditeur Unity, il faut que cette variable soit publique. Ce n'est pas le cas, en mettant "[SerializeField]" au dessus de votre déclaration de variable, cela aura le même effet. Ainsi:

```
[SerializeField]
private int hp_max_;
```

permet à hpmax_ d'être édité depuis l'éditeur malgré sa visibilité private.
https://docs.unity3d.com/ScriptReference/SerializeField.html
pour en savoir plus.

3.3 Unit

Finie l'abstraction, on entre dans le concret. Nous allons créer la classe Unit qui représentera les unités. Et là, autant vous dire que ça va être funky. Pour faire une unité nous avons besoin de PLEIN de stats.

- Une variable protected de vitesse de déplacement
- Une variable protected de dégâts
- Une variable protected de temps de recharge d'attaque
- Une variable privé pour stocker le temps de la prochaine attaque
- Une variable d'armure
- Une variable de portée
- Potentiellement ce que vous voulez
- Une méthode public de déplacement vers un point
- Une méthode public d'attaque d'une cible

```
class Unit : Entity
{
    [SerializeField]
    protected float move_speed_;
    [SerializeField]
    protected int dmg_;
    [SerializeField]
    protected float atk_cd_;
    private float next_atk_;
    [SerializeField]
```

```
    protected int armor_value_;
    [SerializeField]
    protected float range_;
    //...

    public void move_to(Vector3 target){}

    public void attack_target(Life_entity entity){}

}
```

Avec ça on devrait être pas mal... Quoi que. La question se pose, comment "asservir" notre unité, comme dirait les gars d'Evolutech? Autrement dit, comment dire à une unité "Va là-bas mais te prend pas les murs stp". C'est un problème complexe que l'on appelle le Pathfinding. Plusieurs possibilité existent mais pour des sups comme vous, on va rester simple. Un component Unity existe pour pouvoir faire ça et vous évite de faire le programme du S4 tout de suite, Junior serait déçue de ne rien vous apprendre (déjà que vous restez à vj, un échec en soit). Du coup, je vous invite à lire la doc du NavMeshAgent et son tutoriel dédié sur le site officiel:

https://docs.unity3d.com/Manual/nav-HowTos.html
https://unity3d.com/fr/learn/tutorials/s/navigation

"En fait c'est un flemmard il veut pas nous dire comment ça marche" Alors déjà oui, parce que j'écris ces lignes à la place de ~~jouer~~ réviser mes rattrapages, mais aussi que vous devriez vous entraîner à lire la doc ou comprendre les

tuto qui sont PAS DU TOUT complets 9 fois sur 10.
Une fois que vous avez compris comment marche le navmeshagent et comment bake votre carte, vous devriez arriver à créer une méthode qui permet de déplacer et en vérifiant votre portée vous devriez pouvoir attaquer.
Vous en avez probablement marre de coder et de ne pas pouvoir jouer. Ouais. Je comprends. C'est juste que avant, je commençais par la caméra, mais c'est un peu plus chaud, donc tout le monde abandonnait. Là vous pouvez pas tester, mais j'ose espérer, si vous lisez ces lignes, que vous n'avez pas encore abandonné.

4 Camera

Bon, là vous pouvez pas tester votre code "proprement" donc on va rapidement créer de quoi créer cette caméra et pouvoir jouer un petit peu avec ces unités. Nous allons créer la classe RTS_camera qui héritera de Monobehaviour. Dans cette classe il y aura finalement assez peu de variables et surtout beaucoup de méthode.

```
class RTS_camera : Monobehaviour
{
    [SerializeField]
    protected int border_size_;
    [SerializeField]
    protected float move_speed_;

    private GameObject selected_object_;
    private Unit selected_unit_;

    private void move_by_boundaries();
    private void select_unit();
    private void move_unit();
}
```

J'aimerais bien vous laisser vous débrouiller mais il va falloir introduire quelque notions importantes. Tout d'abord la gestion des Input. C'est bien beau un jeu, mais si vous pouvez pas jouer, c'est moins fun. Tout se passe dans la classe Input qui contient quelque méthode statiques intéressantes (donc n'initialisez pas d'objet Input s'il vous plaît). Il faut aussi rapidement parler des "axes" dans unity. Les axes sont... des axes, influencés par deux touches, et servent d'Input de base personnalisable au lancement (sur le petit

launcher Unity avant de lancer le jeu). Pour ajouter ou changer ces axes, allez dans Edit/Project Setting/Input. Pour pouvoir appeler vos axes, rien de plus simple:

```
float GetAxis(string axisName);
```

Le float renvoyé varie entre -1 et 1. Si vos touches peuvent osciller entre différentes valeurs positives ou négatives, comme un stick par exemple, les valeurs peuvent être dans l'intervalle. Si ce sont des touches binaires, comme des touches de clavier, la valeur sera 0 si aucune touche n'est appuyeé, -1 si la touche négative est appuyée, et 1 pour la positive. Si vous appuyez sur les deux, ça dépend un peu de la situation. Je vous invite à tester.
L'objet Input vous permet aussi d'avoir plein d'autres input, je vous propose donc:

https://docs.unity3d.com/ScriptReference/Input.html

Les dernière choses dont vous aurez besoin pour pouvoir faire cette Camera sont:

- De quoi déplacer votre Camera
  https://docs.unity3d.com/ScriptReference/Transform.html
- De quoi récupérer la position de votre souris, je vous laisse chercher dans Input
- De quoi récupérer un component sur un GameObject.

Je vais vous donner le dernier parce que c'est un peu dur

```
GameObject g = GameObject.Find("Arthas");
Unit u = g.getComponent<Unit>();
```

Il vous suffit de remplacer Unit par la classe que vous voulez récupérer sur votre objet et bien sur changer le type de la variable. Il est à noter que cette méthode renvoie null si le component n'est pas trouvable sur l'objet. Faites donc attention à toujours vérifier si la valeur récupérée est valide (!= null)

Votre Taffe c'est maintenant de faire une camera comme sur Warcraft 3 (ou Lol pour les incultes)

- La camera bouge sur les touches fléchées
- La camera bouge si la souris atteint un bord de l'écran, utilisez border_size_ pour augmenter ou réduire la taille des bords
- Click gauche sur le sol ou le décor ne fait rien, click gauche sur une unité la sélectionne
- Click droit sur le sol ou une unité déplace l'unité sélectionné vers le point clické rien sinon
- BONUS: faites un carré de sélection
- BONUS: faites que vos unités suivent une autre unité et ne sauvegarde pas que sa position au moment du click
- ULTRA BONUS: faites des mouvements en formation. Askip Je (Kenan Lejosne) offre une canette à celui qui arrive à faire la formation en carré et la formation en quinconce de Age of Empire 2. (Offre réservée aux sup, soumise à conditions)

4.1 Déplacement aux flèches

Pour pouvoir déplacer votre camera avec vos flèches de clavier, il va vous falloir tout d'abord créer des axes, comme expliqué plus haut, ou alors réutiliser un axe existant. Ensuite vous devrez, dans Update, vérifier ces axes à chaque frame, et déplacer la camera en fonction de la valeur de retour de getAxis. Pour le déplacement, la fonction Translate de transform devrait vous être utile. N'oubliez pas d'orienter votre camera vers le bas pour avoir la vue de haut d'un RTS.

4.2 Déplacement à la souris

Un peu moins trivial que le précédent, vous devez ici vérifier que la souris se trouve sur les bords. La vérification de la présence de la souris sur un bord peut être exprimé :

$$\begin{cases} x \in [0, border\_size\_] \cup [Screen.width - border\_size\_, Screen.width] \\ y \in [0, border\_size\_] \cup [Screen.height - border\_size\_, Screen.height] \end{cases}$$

Vous pouvez utiliser Screen.height et Screen.width. pour avoir les dimensions de votre écran dans vos scripts. Vous pouvez donc créer une méthode qui renverra si oui ou non vous êtes sur le bord de l'écran. Vous pouvez soit renvoyer un bool, soit renvoyer un Vector3 que vous utiliserez directement pour la translation. Le choix est votre.

```
private bool is_in_borders(float x, float y);
private Vector3 is_in_borders(float x, float y);
```

Une fois que ceci est fait, il vous suffit de vous déplacer selon vos résultats. Et normalement vous devriez pouvoir déplacer votre camera. Pour information,

une fois que vous avez clické sur le bouton play, vous devez clicker une fois dans l'écran de jeu pour que votre souris soit prise en compte, de la même manière que votre souris n'a pas d'effet sur un jeu en pleine écran fenêtré si vous ne clickez pas dessus.

4.3 Sélection d'une unité

Il est temps d'intéragir un petit peu avec votre jeu. Il va vous falloir permettre à votre Camera de récupérer une unité en clickant dessus. Pour ce faire vous aurez très probablement besoin du bout de code suivant

```
RaycastHit hit;
Ray ray = camera.ScreenPointToRay(Input.mousePosition);

if (Physics.Raycast(ray, out hit))
{
}
```

Dans le if vous pouvez utiliser la variable hit pour récupérer des informations sur ce que votre Raycast a touché. Si ce que vous avez touché est une unité, stockez là, sinon ne faites rien. Aussi simple que ça. Si vous ne voyez pas comment différencier le décor et vos unités, j'aimerais attirer votre attention sur les tag, les tags sont des strings qui permettent de marquer des objets (au hasard, marquer vos unités avec quelque chose et votre décor avec autre chose). Ces tags s'editent dans Edit/Project Setting/Tags and Layers. Et je vous invite à les utiliser. Toutefois, je vous mets en garde:

N'UTILISEZ PAS == POUR COMPARER VOS TAGS

utiliser plutot object.compareTag(string tag). cette fonction va comparer vos

tags avec leurs id respectifs. Et c'est BEAUCOUP plus rapide. Donc faites moi plaisir, et ne finissez pas comme pubg et leurs performances d'epitech.

4.4 Déplacement d'unité

Maintenant que vous avez récupéré une unité, il va falloir récupérer son script d'unité avec le bout de code donné plus haut. Une fois que vous avez ce script, lorsque vous faites click droit, vous pouvez déplacer votre unité en utilisant sa fonction move_to avec les coordonnées de votre RaycastHit. Normalement, si vous avez tout fait bien, cela se fait vite

4.5 Quelque tips

Pour rappel, Update est appelé à chaque frame, évitez de faire des boucles dedans sauf si vous savez vraiment ce que vous faites. C'est une étape difficile donc n'hésitez pas à poser des questions. A la fin de cette étape vous devriez pouvoir créer une unité depuis l'éditeur et la déplacer en jeu.

5 Héros et Combat

5.1 Combat

Super, vous avez des unités qui bougent. C'est parfait. Mais maintenant, DU SANG POUR LE DIEU DU SANG, DES CRANES POUR LE TRONE DE CRANE. On va faire que les gens puissent résoudre leurs problème de la manière la plus humaine possible: la baston. Pour se faire, nous allons créer un truc pratique: des factions. Vous pouvez vous y mettre tout de suite, ajoutez un champ faction dans votre Entity. Le type est libre, un int c'est pratique ça permet de faire plein d'équipe, mais faites comme vous le sentez. Bon maintenant, les choses sérieuses commencent. Il va vous falloir trouver un moyen de faire un mouvement offensif. En effet quand vous faites un click droit sur une unité énemie, vous devez non seulement la poursuivre, mais aussi l'attaquer lorsque vous le pouvez. Pour cela, je vous propose un petit pseudo code

```
Procedure move_to_attack:
Toutes les n secondes:
     move_to(target.transform.position)
    Si range(this, target) < range :
     stop_move()
        attack()
    fin si
```

C'est très globalement ce que vous devez faire, bien évidement, target est de type GameObject et vous devez faire votre implem pour le reste. N'oubliez pas de mettre un temps de recharge sur l'attaque. Encore un petit pseudo code pour vous permettre de faire un temps de recharge simple

```
Si Time.time > next_attack:
deal_dmg();
    next_attack = Time.time + attack_cd;
fin si
```

C'est assez simple mais ça vous permet de faire pas mal de chose simple. Time.time vous renvoit le temps en seconde depuis le lancement du jeu en float pour information. Vous devriez normalement pouvoir vous en tirer avec ça. faites que vos unité puissent attaquer vos adversaires. A priori, vous devriez aussi faire que la camera à une faction et l'empêche de sélectionner des unité qui ne sont pas de cette faction.

5.2 Héros

Il est temps de créer la source de l'épidémie des MOBA: le système de héros. En soit, les héros sont de simples unités, à l'exception qu'ils gagnent de l'expérience et ont souvent des sorts. Pour ce tp, nous nous limiterons à avoir de l'expérience, faites des sorts si cela vous amuse.
Pour cela, nous allons simplement une classe héritant de Unit qui s'appellera Hero:

```
class Hero : Unit
{
    [SerializedField]
    protected int xp_;

    [SerializedField]
    protected int level_;

    protected void update_level(){}
```

```
    protected void update_stat(){}
}
```

Avec cette classe minimaliste, vous pouvez désormais gérer de l'xp. Mais il va aussi vous falloir ajouter une valeur d'xp stocké sur vos unités pour savoir combien ils rapporteront lorsque vous les tuerez. Vous pouvez ainsi récupérer son xp à sa mort et update votre niveau et vos stats. Pour le coup l'implémentation pour récupérer l'xp est assez libre. Sachant que vous avez accès à la classe, vous pourriez manuellement vérifier ses hp. Ou alors avoir une valeur de retour sur votre fonction qui applique les dommages qui renverrait l'xp gagné (cela vous permet de faire un système de gain d'xp sur les attaques si cela vous intéresse). enfin bref, cette partie est assez libre, car c'est principalement du gameplay.

Si vous voulez des progression de statistiques différentes, vous pouvez passer update_stat en virtual et créer une classe pour chacun de vos héros différent (ça sera de toute façon pratique si vous faites des sorts) et override la méthode pour avoir votre progression spécialisé.

De la même manière, si vous voulez faire des sorts, faites le probablement dans Entity avec un ensemble méthodes virtuels qui permetteront de définir dans les sous classes le comportement des différents sorts

```
class Entity : GameObject
{
    public virtual void qSpell(){}

    public virtual void wSpell(){}
    //...
}
```

```
class Hero : Unit
{
        public override void qSpell(){}

    public override void wSpell(){}
    //...
}
```

Vous pouvez ainsi appelez vos touches dans l'Update de votre Unit mais définir le comportement dans vos sous-classe.

6 Building

Vos gars se foutent désormais sur la gueule. et ça c'est cool mine de rien. Rien que là vous avez une bonne première soutenance, et si vous êtes encore là on va peut être arriver à une deuxième soutenance (sauf que y'a pas de réseau ni vraiment de contenu donc bon, bougez votre cul un peu). On va parler batiments, ingénieur c'est bien, ingénieur dans le batiment c'est mieux. Mais alors, qu'est-ce que c'est que quoi-ce est un batiment? C'est une entité qui a des PV et qui peut créer des unités ou au moins faire des updates. Pour la première partie, c'est assez facile, vos Life_entity font déjà un bon boulot. Et maintenant pour la liste d'unité? Bon, on va être honnête, c'est un domaine un peu piège. Cette liste comporte des éléments de GUI, interagis avec les ressources et un points de spawn. Ca fait beaucoup comme ça. Mais on va faire ce qu'on peut.

```
public struct Spawn_info
{
        public GameObject to_spawn;
    public float spawn_time;
    public int ressource;
}
public class Building : Entity
{
    [SerializedField]
    protected List<Spawn_info> spawnables_;

    protected Vector3 rally_point_;

    public void Spawn(int _id){}
```

```
    public void change_rally(Vector 3 _pts){}

}
```

Je ne vous présente plus *El famoso* List de C#, comme vous pouvez potentiellement faire spawn plusieurs unités et qu'elles ont chacunes leur temps de spawn, je vous propose de stocker les infos relatives à votre objet à spawn dans une ~~classe~~ struct, comme ça si vous avez à changer le nombre d'info (genre augmenter le nombre de ressource) vous êtes clean. Et du coup dans votre super Buildings vous avez une classe d'info. Simple, basique. Vous pouvez presque tout faire en fait! Ah non, vous ne savez pas comment faire spawn un objet.

```
public static Object Instantiate(Object original,
   Vector3 position, Quaternion rotation);
```

C'est assez simple, un objet à spawn, une position et une rotation. Alors pour la rotation... Les Quaternions c'est un peu compliqué.

In mathematics, the quaternions are a number system that extends the complex numbers

Voilà, donc on va faire sans pour le cadre de ce tp en utilisant gameObject.transform.rotation qui vous permet de récupérer la rotation de votre objet, et ainsi la donner à votre création. Pour tester, comme c'est normalement sur l'interface vous avez deux choix:

- Vous avez fait les fonctions virtuels de la partie précédente et vous n'avez qu'à override
- Vous ne l'avez pas fait et dans ce cas là faites comme vous le sentez.

En résumé vous devez faire les comportements suivant:

- Faire que les bâtiments soient sélectionnables
- Faire que click droit quand sélectionner replace le point de ralliement au points clické
- Possibilité de faire spawn des unités en sélectionnant le bâtiment. Les unités vont par défaut au point de ralliement

coman on fé pour renplire la listte? C'est effectivement un problème. Mais ne vous inquiétez pas, j'ai la solution. ~~Demerdez vous.~~ Vous allez devoir utiliser la fonction Start(), créer des prefab et utiliser Ressources.Load(). Comme vous avez rien compris, je la refait doucement. Jusqu'ici nous avons utiliser l'une des deux méthodes générés de base dans vos script. Il est temps d'utiliser la deuxième. Start() est une fonction d'initialisation. Elle permet de charger ce dont vous avez besoin au tout début, avant la première frame. Un peu comme des objets qui ne changeraient jamais, comme une liste d'objet à spawn *wink wink*. Pour charger des ressources, vous pouvez utiliser la fonction Ressources.Load() qui va aller chercher dans votre projet un objet à load, je vous invite à lire cette page de doc plutot courte pour en savoir plus:
https://docs.unity3d.com/ScriptReference/Resources.Load.html

Les Assets C'est bien vous savez comment charger des assets mais vous n'en avez pas. Rien de plus simple, créez un objet sur votre scène, drag and drop votre objet depuis votre hierarchy vers votre explorateur unity et bim. un asset. Libre à vous d'attacher ce que vous voulez sur vos assets (un script Unit pourrait être cool, je dis ça comme ça hein). Et mettre un nom cool,

vous pouvez maintenant le charger. Votre Start devrait ressembler à quelque chose comme ça

```
public void Start()
{
        to_spawn = new List<spawn_info>();
        to_spawn.Add(new
           spawn_info(Ressources.Load("Thrall"), 20,
           100));
    to_spawn.Add(new
       spawn_info(Ressources.Load("Rengar"), 40, 150));
    to_spawn.Add(new
       spawn_info(Ressources.Load("Grom"), 200, 1000));
    //On setup aussi le point de ralliement initial,
    //j'ai pas trop d'idee donc:
    rally_point_ = gameObject.transform.position;
    rally_point_.x += 2;
}
```

Et bim, vous pouvez faire pas mal de truc maintenant. Sauf que vous ne pouvez pas encore tester vos ressources. Bah non il y a pas encore de système de ressources... *Pour l'instant.*

7 Ressources

Bon les enfants. Il va falloir que je vous fasse part d'un secret, c'est pas si facile que ça de faire les ressources. Il va y avoir des trucs un peu cool (pas trop mais un peu), donc accrochez vous, vous y êtes presque. Et si vous en avez marre, VJN a des crêpes ou alors vous pouvez aller faire de la pâte à modeler sur blender.

7.1 Le singleton

Les puristes me demanderont sans doute comment on a pu tenir tout se temps sans un singleton. Mais moi je m'en balance. Un singleton c'est quoi? C'est un design pattern, en gros c'est un truc qui sert pour plein de problèmes et que vous pouvez réutilisez dès que ce problème se présente. Ici quel est le problème? En fait le problème c'est qu'il faut que votre jeu soit géré par quelque chose de *supérieur*, en d'autre terme il faut une classe au dessus du tumulte qui va gérer, par exemple, les ressources des joueurs. Quand je dis gérer, je sous entends avoir la variable qui s'en occupe. Ici on ne parle que d'une variable mais vous verrez que vous pourrez en avoir besoin de plus. Mais il faut que cette classe **n'ait qu'une seule instance et qu'elle soit accessible de partout**. Ce texte en gras est plus ou moins votre cas d'utilisation type d'un singleton. En code ça donne quoi? C'est assez simple, il faut que votre classe ait un membre de son propre type en static et que tout ses constructeurs soit privés ou protected. Il vous faut finalement une méthode public static qui vous donnera l'instance unique.

```
public class Game_manager
{
        public static Game_manager Instance()
```

```
    {
        if (instance == null)
                instance = new Game_manager();
        return instance;
    }

        private Game_manager(){}
        private static Game_manager instance;
}
```

Aussi simplement que ça. Si vous voulez vous renseigner sur le pattern, je vous invite à lire la page wiki, parce que ce n'est pas le sujet du TP. Toutefois vous pouvez désormais ajouter ce que vous voulez dans la classe ou modifier le constructeur. Si vous avez besoin d'accéder à votre instance de Game_manager depuis ailleurs il vous suffit d'appeler la fonction statique:

```
Game_manager gm = Game_manager.Instance();
```

Bon, on va pouvoir se mettre au travail. Maintenant que vous avez un singleton qui déchire vous allez pouvoir faire plusieurs chose:

- Mettre une variable de ressource dans votre Game Manager
- Vérifier que vous avez suffisamment de ressources dans building avant de lancer le spawn

Bon c'est cool, mais comment on choppe de l'oseille maintenant? Bah en fait, vous pouvez faire de plein de méthode différentes en soit. Vous pourriez simplement faire que vous en gagnez périodiquement. Ca pourrait être sympa. Mais sinon vous pourriez créer une sous classe d'entité genre filon qui permet

à un type d'unité spécial de récolter la ressource. Ou même mieux, de la récolter pour ensuite aller la ramener dans un bâtiment, lui-même spécial, et enfin gagner le fameux gold. Comme on arrive sur la fin du TP (parce que la GUI c'est pas vraiment du code sur Unity), c'est un peu le turbo bonus.

Le premier qui me fait un système de ressources comme warcraft 3 (donc deux ressources distinctes) prend une canette. Je veux les bâtiments spéciaux, hotel de ville pour les deux et une scierie pour seulement une des deux ressources.

8 GUI

Vous êtes encore en vie? Du coup vous avez un jeu, c'est cool. Vous avez des trucs qui fonctionnent, c'est cool. Mais vous savez ce qui serait encore PLUS cool? De l'interface utilisateur. Parce que actuellement c'est un peu compliqué de comprendre ce qui se passe à l'écran. On va donc partir sur une interface simple, libre à vous de faire plus.

- Une interface de status d'unité affichant la vie et attaques/dmg
- Une interface qui permet de créer les unités depuis les batiments
- Une interface qui permet d'afficher les ressource

Pour faire cela, tout d'abord je vous invite à lire la doc, parce que je ne suis pas expert en UI sur unity
https://unity3d.com/fr/learn/tutorials/s/user-interface-ui
Une fois que cela est fait, on va parler de l'interaction entre votre ui et votre jeu. Un truc bien long et relou qui est souvent sous estimé dans les grosses prod. Ici on va rester simple et efficace. Point par point je vais vous dire comment récupérer ces infos.

8.1 Info d'unité

Commençons par ces infos, déjà, qui est concerné? Les unités, les héros et les bâtiments. La classe mère commune est donc Life_entity. En gros, que voulons nous? Il nous faut la vie mais aussi d'autres stats et l'on veut que chacune de nos classes puissent mettre des infos différentes. Il y a plusieurs méthode mais je vous propose la suivante: Créer une struct qui stockera simplement un ensemble d'info qui sera lu par votre UI. il suffira d'implémenter

une méthode qui remplira cette struct et overloader cette méthode dans vos sous-classe et c'est gagné:

```
struct Ui_info
{
        int hp;
    int defense;
    //other things you might want
}
class Life_entity : Entity
{
        public virtual ui_info getInfos(){}
}
```

Avec ça vous pouvez efficacement faire des requêtes d'information, au hasard en passant par votre caméra qui sélectionne votre objet. Si vous voulez des comportement spéciaux vous pouvez toujours override dans vos classes filles. Petit pro-tips, si vous voulez ajouter du comportement (par exemple, votre héro affiche la même chose que vos life_entity mais ajoute ses stats), overridez votre methode et appelé sa méthode mère:

```
class Hero : Life_entity
{
        public override ui_info getInfos()
    {
        ui_info stdInfo = base.getInfos();
        //Do something more

        return stdInfo;
    }
}
```

Et là c'est pas beau? Bordel oui ce l'est. Et en plus c'est informatiquement pas *trop* dégueu, vous vous rendez compte? Du code propre en sup.

8.2 Les boutons de création

Je ne sais pas quoi vous dire. En réalité vous savez déjà créer un bouton en UI si vous avez suivi la doc, vous avez une fonction qui permet de faire spawn des unité... Faites le lien.

8.3 Afficher vos ressources

On y est presque. Et vous savez quoi? Vous savez le faire. Vous venez de faire quelque chose qui permet de récupérer des infos via une struct. Et pour vos ressources? C'est pareil. Créez une struct, créez une méthode qui récupère ces infos et affichez les. Ca devrait le faire je pense.

9 Conclusion

Si vous lisez ces lignes, c'est soit que vous avez rien branlé, soit que vous l'avez fait. Et c'est cool. J'ai rushé ce tp pendant cette semaine donc il se peut que ce ne soit ultra fun des fois, je m'en excuse, il reste 40k fois mieux que le précédent qui a mis en PLS la promo. Si vous avez des questions sur unity, blender, la vie la mort, belzebub ou le bien suprême de l'empire T'au mon mail est au début du TP.

10 Sources

Page de garde : http://wallpaperswa.com/Space/Moon/fatestay_night_unlimited_blade_works_emiya_shirou_typemoon_fate_series_147416

Just because you're correct doesn't mean you're right