

PROJET ASTEROID BELT

Tuto Entity, Component et Services

Introduction

Ce tuto va vous servir à comprendre comment implémenter un système Entity/Component basique. Vous verrez comment implémenter vos composants avec un Object Pool, et comment découper votre code sous forme de Services. Nous prendrons l'exemple du GraphicsService pour démontrer l'utilisation. Nous verrons également des techniques avancées pour facilement ajouter un nouveau service, et stocker vos composants dans vos entités.

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 : Créer une Entity simple

Le système Entity/Component va être au cœur de votre système de jeu.

C'est un élément critique et il doit bien fonctionner. Alors repartons de zéro.

Selon vous, que devez-vous mettre dans une entité ?

- Une orientation ? Une position ?
- Un sprite ?
- Des points de vie ?

La réponse est : rien de tout ça... ou presque. Prenez par exemple :

- Un objet « abstrait » comme un spawner d'ennemis : il n'a pas besoin d'orientation, par contre il a besoin d'une position.
- N'importe quel objet invisible n'a pas besoin d'un sprite. Et certains n'ont même pas besoin de position.
- Tout ce qui n'est ni le player, ni un astéroïde, n'a besoin de points de vie. Un power-up par exemple n'a pas besoin de points de vie.

Autrement dit, dans notre modèle une Entity est un concept abstrait dans lequel il n'y a rien à la base car rien ne nous semble indispensable à « n'importe quel type d'entité ».

L'implémentation en est donc très simple :

```
class Entity
{
};
```

Etape 2 - Brainstorming

Bien sûr, ça ne s'arrête pas là...

Maintenant, il nous faut réfléchir à nos entités en terme de **features**. De quoi ont besoin les entités de notre jeu ? Faisons une première passe et énonçons des faits :



- Le player et les astéroïdes ont besoin d'un sprite à afficher.
- Le player et les astéroïdes ont besoin de HP.
- Le player et les astéroïdes doivent être capables de jouer un son.
- Les astéroïdes ont tous le même sprite, seule leur transformation (position/rotation/scale) est différente.
- Les powerup ont besoin d'un sprite à afficher, et n'ont pas besoin de HP. Par contre, ils ont besoin de savoir jouer un son (quand on les touche).
- Les powerup, le player et les astéroïdes doivent être capables de vérifier des collisions.
- Les objets dans le background (parallax) ont besoin d'un sprite à afficher, et n'ont pas besoin de HP. Ils n'ont pas besoin de savoir jouer un son. Ils n'ont pas besoin de savoir vérifier une collision.
- Nous allons peut-être avoir besoin de spawners d'astéroïdes, et peut-être de spawner de background objects. Un spawner est un objet qui n'a besoin ni de sprite, ni de HP, mais a besoin d'une position.

Nous pourrions continuer longtemps comme ça...

Ce que je veux vous faire comprendre c'est que :

- Certaines parties de la logique sont **inhérentes** au **type** d'une entité : par exemple :
 - o le player va vouloir vérifier si il entre en collision avec un astéroïde
 - o les astéroïdes vont vouloir vérifier si ils sont en collision avec une bullet
 - o les power-ups vont vouloir vérifier si ils sont en collision avec le player
- Et certaines parties de la logique sont **partagées**, c'est-à-dire qu'on pourrait coder une brique réutilisable par chacun des types d'entité car fondamentalement, leur besoin est le même :
 - o Afficher un sprite.
 - o Jouer un son.
 - o Tenir un compte d'HP.

Sans trop non plus aller dans le détail, nous pouvons donc déjà identifier trois composants réutilisables potentiels :

- Un composant qui se charge de l'affichage d'un sprite
- Un composant qui se charge de jouer un son
- Un composant qui permet d'avoir des HP.

Commençons par le plus simple : celui des HP.

Etape 3 – Health Component

Considérons que nos HP sont des float.

On peut donc commencer à écrire un petit component :

- Une class **HealthComponent** dont le constructeur prendra un float en paramètre
- Qui stockera un float m_hp initialisé dans le constructeur
- Qui aura des méthodes GetHP() et SetHP(). Attention les HP ne peuvent pas être négatifs.

Bon, ça c'était facile. Maintenant il faut s'en servir.

On ne peut pas le mettre dans l'Entity car on l'a vu, toutes les entités n'ont pas besoin d'HP.

Par contre, on a vu qu'au moins le player et les astéroïdes ont des HP.

On pourrait donc commencer par créer une class d'Entité qui dérive d'Entity, que nous appellerions **HealableEntity**, et qui utiliserait ce Health Component.

- Codez une class **HealableEntity** qui dérive d'Entity, et qui possède un membre m_hpCpnt de type HealthComponent en **protected**.



OK ! Nous voilà avec notre premier type d'entité !

Etape 4 – Player et Asteroid

Ensuite, rappelons-nous ce que nous avons dit plus haut :

- le player va vouloir vérifier si il entre en collision avec un astéroïde
- les astéroïdes vont vouloir vérifier si ils sont en collision avec une bullet

Nous avons donc là deux logiques bien distinctes. Je ne vois pas d'autre solution que de séparer les deux, car elles n'ont rien à voir l'une avec l'autre.

- Créez une classe **PlayerEntity** et une classe **AsteroidEntity** qui héritent toutes deux de **HealableEntity**.
- Elles auront toutes deux une fonction Update qui prend un float deltaTime en paramètre. Nous nous occuperons de les coder plus tard.

Super ! Nous avons donc maintenant deux entités Player et Asteroid qui partagent toutes les deux le même type de component d'HP ! C'est bien la preuve que les components sont une façon très pratique de partager :

- À la fois des structures de données (les hp)
- Et de la logique (les fonctions du component Health)

Poursuivons nos efforts...

Etape 5 – Collision component ?

Le powerup est un objet intéressant car :

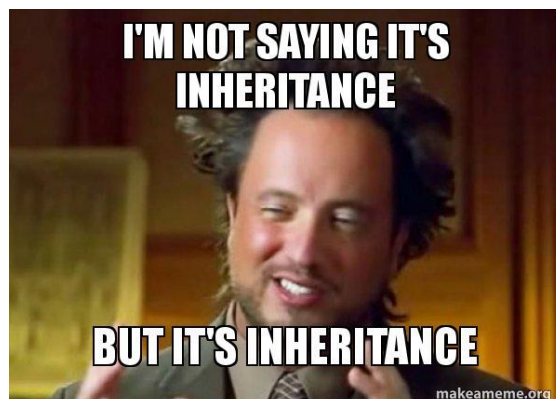
- o les power-ups vont vouloir vérifier si ils sont en collision avec le player

C'est donc une entité gameplay, mais qui n'a pas d'HP. Donc, elle n'héritera pas de HealableEntity. Pourtant, elle aura besoin de vérifier des collisions avec le joueur...

- Idée : et si on codait un component « **CollisionComponent** » qui s'occuperait de prendre deux entités et de vérifier si les deux collisionnent ?
- Après tout, le player et l'astéroïde aussi s'en serviraient...
- On aurait dans le component une enum class qui dirait quelle type de forme de collision on a (AABB, OBB, cercle...)
- Et pourquoi pas une **union** pour stocker les données de ce type de collision ? Après tout, la même entité n'aura jamais deux types de collision différents, on peut donc réutiliser la même mémoire.
- Et si on codait une classe **CollidableEntity** qui hérite d'Entity et contient un **CollisionComponent** dedans ?

Nous voilà face à un terrible dilemme : ça semble être une bonne idée sur le papier, mais on se rend compte que comme le player et les asteroides ont tous les deux des HP, et ont tous les deux des collisions, mais que le powerup a des collisions mais pas de HP, il faut **décorréliser** les deux. On devrait donc faire hériter Player et Asteroid de HealableEntity et CollidableEntity, et ça, ça crée un héritage en diamant sur Entity.

Et l'héritage multiple, en diamant qui plus est, on aime pas ça, et on en veut pas. Du tout.



Il va nous falloir avoir une autre idée... Nous reviendrons dessus un peu plus tard.

Etape 6 – GraphicComponent ?

Il y a une chose que le powerup, le player et l'astéroïde ont en commun : ils ont tous besoin d'afficher un sprite.

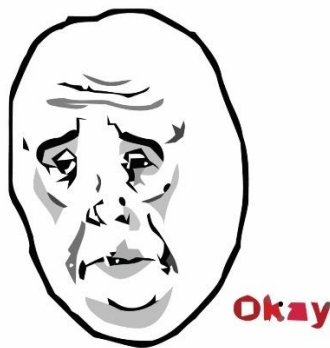
Je pense que cela remplirait bien le mandat d'un GraphicComponent, par exemple :

- Coder une classe GraphicComponent qui prend en paramètre constructeur un `sf::Sprite` *
- Ce GraphicComponent va stocker :
- Un `sf::Sprite` qu'on va afficher plus tard.
- Une transformation (donc position, orientation et scale) à appliquer à ce sprite.

C'est cool ! Comme ça ça va nous permettre d'afficher un sprite à l'écran.

Mais encore une fois, le même problème s'oppose à nous. Devrions-nous coder une classe `RenderableEntity` ? Bien sûr que non.

Il semble que nous soyons dans une impasse. 😞



Etape 7 – Retour Sur Le Rendering Service

Souvenons-nous du cours. Qu'est-ce qui était si cool avec cet exemple ?

```
class RenderingService
{
public:
    void UpdateRender(float dt)
    {
        for (const auto& renderCpnt : m_components)
        {
            // ...
        }

        // draw frame ...
    }

private:
    std::vector< RenderComponent > m_components;
};
```

- Le fait que les composants soient tous les uns à côté des autres en mémoire (cache-friendly !)
- Le fait qu'on y accède de manière séquentielle (cache-friendly !)
- Le fait qu'ils puissent être réutilisés (si on utilise un Object Pool)

Nous aimerions reproduire cette architecture dans notre projet. Malheureusement, nous sommes un peu partis dans la mauvaise direction :

- Nous avons stocké nos composants directement dans les entités, ce qui potentiellement casse le fait que les composants soient tous les uns à côté des autres en mémoire.
- De même, comme ils sont dans des entités différentes, on a aucune garantie d'y accéder « les uns derrière les autres ».
- Enfin, on n'a absolument aucune forme d'Object Pool. Quand on va détruire notre entité, tout sera jeté à la poubelle. Aucun recyclage.

Procédons par étapes. Le plus simple est peut-être de commencer par l'Object Pool.

Etape 8 – L'Object Pool

Pour le moment oublions toutes ces histoires d'entité et component et focusons-nous sur l'object pool.

Nous allons tenter de coder un object pool dans les règles de l'art. Ce ne sera peut-être pas le meilleur object pool de l'univers mais au moins il marchera.

De quoi avons-nous donc besoin pour coder un Object Pool ?

- D'une structure de données **contigüe**, mais qu'on essaiera de **ne pas faire réallouer**. En effet, le but de l'object pool est de donner à l'extérieur des références vers les objets à l'intérieur du pool (pour justement **éviter les copies** et la **fragmentation mémoire**). Qu'arriverait-il si le conteneur, faute de place, venait à **réallouer** toute sa zone mémoire ? Toutes les références que nous avons donné auparavant deviendraient fuckées (on dit plutôt « **invalidées** » pour être plus poli), car elles pointeraient désormais sur de la mémoire **libérée** !! (délivrée ?)
- Nous avons besoin de ce qu'on appelle une « **free list** » pour savoir quelles sont les cases dans le tableau qui sont libres ou pas. En effet, on n'a aucun contrôle sur l'ordre de « get un nouvel objet » et « retour d'un objet détruit ». Il peut donc très vite se former des « **trous** » dans le tableau...
- Vue d'artiste d'un object pool après quelques utilisations :

LIBRE	OCCUPE	LIBRE	LIBRE	LIBRE	OCCUPE	OCCUPE	LIBRE	OCCUPE	LIBRE
-------	--------	-------	-------	-------	--------	--------	-------	--------	-------

Hors de question de reparcourir toute la liste pour trouver le premier élément « LIBRE ». Car dans le pire des cas, ce serait de complexité **O(n)** pour une opération qui devrait être triviale. On aimerait du **O(1)** ici ! Alors, on préfère sacrifier un peu de mémoire pour garder une trace des « index » libres que l'on peut utiliser dans le tableau.

Voici donc les questions auxquelles vous devez répondre :

- Quelle structure de données peut-on utiliser pour le pool ? Il faut que les données soient **contigües**. Dans notre cas, on peut se contenter d'allouer une taille fixe d'objets relativement grande (128, par exemple), et on avisera après si un jour on dépasse cette limite. Mais **attention** : on ne veut pas **construire** des objets, on veut juste **allouer la mémoire nécessaire**. Il y a une différence ! Les objets seront créés quand on demandera un nouvel objet à l'Object Pool.
- Quelle structure de données peut-on utiliser pour la free_list ? Rappel : la free list n'est qu'une liste **d'index** (on pense à des size_t) contenant l'index de toutes les cases « **libres** » dans le tableau d'objets (le pool). Mais ce n'est pas tout : on aimerait que la free list nous aide à rester **cache-friendly**. Imaginez : on alloue 30 objets, puis on en libère un, et on en réalloue un autre tout de suite après. L'idéal serait qu'elle nous renvoie **le même index que l'objet que nous venons de détruire** : avec un peu de chance, cette adresse mémoire est encore en cache. Comment faire ? N'est-ce pas un comportement « dernier arrivé, premier sorti » ? (Last In, First Out, ou LIFO) ? **A-t-on vu une structure de données LIFO dans les cours** (sur les containers) ?

- Vous justifierez vos choix de containers dans le Document Technique.

Une fois que vous avez choisi vos conteneurs, vous devrez les implémenter en variables membres d'un type template **ObjectPool** qui prendra deux paramètres template :

- Un type T
- Un « **non-type template parameter** » `size_t PoolSize` qui indique la taille du pool d'objets.

De plus, on voudra vérifier avec un **static_assert** que **PoolSize** est > 0 , car ça n'a pas de sens de créer un **ObjectPool** vide.

Enfin, on aura besoin de coder deux fonctions publiques :

- **GetNew()**, qui va aller chercher dans la free list le premier index vers une case libre et allouer, grâce au **placement new**, un nouvel élément dans ce tableau. On souhaite également que **GetNew()** prenne des **arguments template variadiques** afin de pouvoir passer n'importe quels paramètres au constructeur de notre objet et ce, quel que soit le type T.
- **Recycle(T& object, size_t index)** qui prendra un objet à recycler et son index dans le tableau, et qui détruira l'objet en appelant explicitement le destructeur de T et remplacera l'index en question dans la free list.
- Aussi, dans le constructeur vous n'oublierez pas de réserver la mémoire nécessaire pour le pool, et de remplir la free list d'index qui vont de 0 à **PoolSize-1** (inclus), dans cet ordre.
- Comme je suis sympa : je vous donne **GetNew()** :

```
template <typename... Args> <T>
std::pair<T&, Index> GetNew(Args&&... args)
{
    assert(m_freeList.size() > 0); // if we try to get a new element when the free list is full, it's bad news !!
    // Take the index of first available object in the pool.
    size_t firstAvailableIdx = m_freeList.top();
    m_freeList.pop();

    // Now create this new element using placement new.
    T* elem = m_pool.data() + firstAvailableIdx;
    new (elem) T(std::forward<Args>(args)...);

    // and return it !
    return { *elem, firstAvailableIdx };
}
```

Vous pouvez déjà tester votre object pool avec des cas simples : Voici un cas d'exemple et l'output attendu :

```
class ahquecoucou
{
public:
    ahquecoucou()
    {
        std::cout << "coucou!\n";
    }

    ~ahquecoucou()
    {
        std::cout << "au revoir.\n";
    }
};

{
    ObjectPool<ahquecoucou, 128> optic2000;
    // va nous retourner des paires de <ahquecoucou&, size_t index>
    auto ohmarie = optic2000.GetNew();
    std::cout << "ohmarie index: " << ohmarie.second << '\n';

    auto penitencier = optic2000.GetNew();
    std::cout << "les portes index: " << penitencier.second << '\n';

    auto allumez = optic2000.GetNew();
    std::cout << "le feu index: " << allumez.second << '\n';

    auto souvenirs = optic2000.GetNew();
    std::cout << "souvenirs index: " << souvenirs.second << '\n';

    optic2000.Recycle(souvenirs.first, souvenirs.second);

    auto souvenirs2 = optic2000.GetNew();
    std::cout << "souvenirs souvenirs should be same index: " << souvenirs2.seco
nd << '\n';
}
```

```
coucou!
ohmarie index: 0
coucou!
les portes index: 1
coucou!
le feu index: 2
coucou!
souvenirs index: 3
au revoir.
coucou!
souvenirs souvenirs should be same index: 3
```

Vous remarquez que votre Object Pool n'appelle pas les destructeurs des objets sur son propre destructeur. C'est parce qu'on **suppose** qu'un object pool n'est détruit qu'une fois que tous les éléments à l'intérieur ont été rendus au pool.

Vous pouvez si vous le souhaitez faire un **assert(m_freeList.size() == PoolSize)** ; dans le destructeur de l'Object Pool pour vous assurer que tous les objets ont été retournés au pool lors de sa destruction. Est-ce que c'est grave si les destructeurs ne sont pas appelés ? Ça dépend du type stocké et de la durée de vie de votre pool... En tout cas un pool détruit alors que des objets à lui sont toujours dans la nature, ce n'est pas normal !

[Enfin, en bonus :

Vous pouvez essayer de gérer le cas où l'object pool est plein et qu'on a besoin d'un nouvel objet. Cela sera sûrement à base de `std::list`. Une idée d'implémentation :

- Plutôt que juste avoir une freelist et un array, on aura une `std::list` de ces deux trucs-là.
- Créez un premier node contenant un pool lors de la création de l'Object Pool
- Ensuite quand le pool est plein, allouez un nouveau node et allouez des objets depuis le nouveau node.

Retourner un nouvel objet est assez facile. Mais recycler un objet devient dur. À partir du moment où on commence à avoir plus d'un node de `list`, quand on nous passe un objet à recycler, on ne sait plus de quel node de la liste il provient. Vous devrez soit retourner autre chose qu'un index pour identifier l'objet, soit faire un truc « intelligent » avec l'index, par exemple : on sait que notre index est un `size_t`, hors un `size_t` fait 64 bits, ce qui fait une valeur maximale de `ULLONG_MAX`. C'est beaucoup plus que le nombre d'objets qu'on pourrait avoir. On pourra donc « encoder » dans l'index qu'on retourne deux informations : l'index dans notre array, et l'index de notre node de la linked list. On stockera l'index de l'array dans les premiers 32 bits, et l'index du node dans les autres 32 bits, et on y accédera avec un masque binaire (avec l'opérateur binaire `&`) et des décalages de bits (avec l'opérateur bitshift `>>`). Le recyclage deviendrait donc une opération en $O(k)$ avec k le nombre de nodes dans la liste chaînée de pools.

Fin bonus]

Etape 9 — Rendering Service, le Retour

Forts de notre nouvel Object Pool, vous allez pouvoir implémenter le Graphics Service (aussi appelé Rendering Service).

En gros, votre GraphicsService va contenir un object pool de GraphicsComponent's qu'il va « prêter » à tous ceux qui en ont besoin, en allant les piocher dans l'object pool.

Pour commencer, créez une classe **Component** qui contient un `size_t` index. Notre component va stocker son index dans l'object pool et nous nous en servons au moment de le recycler. Vous aurez besoin d'un getter et d'un setter.

Ensuite, créez un **GraphicComponent** qui hérite du **Component**. Je propose cette implémentation, elle vaut ce qu'elle vaut, vous la contesterez peut-être et vous pouvez vous en éloigner, mais je la justifierai un peu plus tard, dans un autre tuto (celui sur le rendu). Pour le moment je vous propose de « bear with me » :



```
class GraphicsComponent : public Component
{
public:
    GraphicsComponent() = default;

    GraphicsComponent(sf::Sprite* sprite, sf::Vector2f pos, float angle, float scale) :
        m_sprite(sprite), m_pos(pos), m_angle(angle), m_scale(scale)
    {
    }

private:
    sf::Sprite*    m_sprite = nullptr;
    sf::Vector2f  m_pos;
    float          m_angle = 0;
    float          m_scale = 1;
};
```

Cette duplication de la position, de l'angle et du scale va permettre, plus tard, à plusieurs objets de référencer le même sprite, mais de lui donner une transformation différente.

Et comme un `sf::Sprite` au complet fait presque **300 octets**, on préfère stocker un pointeur et 4 floats (pour un total de **24 octets**) plutôt que dupliquer le sprite source.

(Nous devons trouver un moyen de stocker les sprites de manière pérenne... Dans le GraphicsService peut-être ? C'est une question pour une autre fois.)

Vous êtes à présent prêt à construire votre GraphicsService. Pour commencer, vous y mettez :

- Private : un ObjectPool<GraphicComponent, 128> (soyons fous)
- Public : GraphicsComponent* **CreateComponent()** : une fonction qui va venir chercher un nouveau component dans l'object pool, et qui appellera **SetIndex** sur ce component pour y mettre l'index retourné par **GetNew()**.
- Public : void **RecycleComponent**(GraphicsComponent* cpnt) : va renvoyer au recyclage dans l'object pool le component qu'on lui passe en paramètre. On passera à la fonction **Recycle** de l'object pool l'index contenu dans le component.

Nous avons donc un service basique qui est capable de « prêter » et « reprendre » des components qu'on lui emprunte. Vous pouvez tester les fonctions **CreateComponent** et **RecycleComponent** et vérifier qu'elles ne crash pas avant de passer à la suite.

Etape 10 – GameManager

Il est à présent temps d'incorporer votre GraphicsService à votre GameManager si ce n'est pas déjà fait.

Pour le moment votre GameManager sera très simple :

- Il possède un **GraphicsService** en **private**
-
- Il possède une méthode **template T* CreateComponent()** en **public** que nous spécialiseront avec une **explicit template specialization** pour retourner des **GraphicComponent** depuis le **graphics service**.
-
- Une fonction **template DestroyComponent(T* cpnt)** qui va se charger de détruire les components, et qui elle aussi fera une explicit template specialization pour donner au graphics service le component à recycler.
- Optionnellement, vous pouvez refactorer **DestroyComponent** pour qu'elle check si le pointeur est null, et qu'elle appelle une autre fonction template et en private qui sera elle, explicitement spécialisée. Cela permet de ne pas dupliquer inutilement le check de pointeur null.

L'idée est de « **rediriger** » la demande de nouveau component et la destruction de component vers le service approprié en fonction du type de component qu'on nous passe.

Ainsi, une entity sera capable de faire **manager->CreateComponent<GraphicComponent>()**, et le manager transmet la demande au bon service (ici le graphics service).

Voilà par exemple à quoi devrait ressembler CreateComponent :

```
class GameManager
{
public:

    template <typename T>
    T* CreateComponent()
    {
        return CreateComponent<T>();
    }

    template <>
    GraphicsComponent* CreateComponent()
    {
        return m_graphicsService.CreateComponent();
    }
}
```

On remarquera que l'object pool nous permettrait théoriquement de passer des arguments au constructeur, mais que notre interface CreateComponent ne le permet pas. Vous devrez donc initialiser vos composants par le biais de fonctions à part et non par le constructeur.

C'est une limitation technique à cause des templates que j'accepte ; il est possible de faire de la « magie noire » avec les templates pour que ça marche, mais c'est plus compliqué et je tiens à garder le niveau de ces tutos à un niveau pas trop avancé.

Vous pouvez opter pour une autre option qui vous semble plus « adaptée » si vous le souhaitez. Je conseillerais cependant d'attendre au moins la fin du tuto



Etape 11 – Création d'entités

Reprenons notre création d'Entités.

Vous êtes maintenant capable de créer des composants avec un object pool via un service, et de les réutiliser.

Désormais, nous pouvons maintenant créer des entités qui se servent de nos composants.

Rappelez-vous des composants qu'on aimerait créer :

- Un component de rendu (c'est fait !)
- Un component de collision
- Un component de HP
- Un component de mouvement, soyons fous ?

Et les entités qu'on aimerait créer :

- Le player, qui a besoin de collisions, d'HP, de mouvement et de rendu
- Les astéroïdes idem
- Les powerup : pas de HP (par défaut), ni de mouvement si vous voulez les faire statiques
- Les projectiles : pas de HP non plus.

Tout à l'heure nous avons vu que la class Entity est vide. D'après moi ce n'est pas bon. Je pense que n'importe quelle entité est susceptible d'avoir besoin de deux choses :

- avoir un pointeur sur notre **GameManager** car c'est lui qui permettra à chaque entité d'accéder à tous les autres services et donc, à faire sa logique gameplay
- des fonctions **Create** et **DestroyComponent** qui ne font qu'appeler celles du manager, plus pour la « qualité de vie », car n'importe quelle entité est susceptible d'avoir un component. (mais elles sont optionnelles. C'est juste pour éviter de retaper m_manager-> à chaque fois)
- On y mettra aussi une position puisque dans notre jeu, tout a une position, même les choses invisibles... Peut-être que nous ferons quelque chose de mieux par la suite.

On aura donc :

```

class Entity
{
public:
    Entity(GameManager* mgr) :
        m_manager(mgr)
    {
    }

protected:
    template <class T>
    void DestroyComponent(T* cpnt)
    {
        m_manager->DestroyComponent<T>(cpnt);
    }

    template <class T>
    T* CreateComponent()
    {
        return m_manager->CreateComponent<T>();
    }

    GameManager* m_manager = nullptr;

    sf::Vector2f m_pos;
};

```

Ensuite, voici un début de hiérarchie de classe que je vous propose :

- **GameplayEntity** : public Entity
 - Contient un component Movement
 - Contient un component Collision
 - Contient un component Graphics
- **DestroyableEntity** : public GameplayEntity
 - Contient un component HP
- **SpawnerEntity** : public Entity
 - Peut contenir un component Movement si vous voulez faire bouger les spawner à un moment...
 - Sinon : pour le moment il est vide

Les **astéroïdes** et les **players** seront des **DestroyableEntity** alors que les **projectiles** et les **powerups** seront plutôt des **GameplayEntity**. Voici par exemple un exemple d'implémentation de la GameplayEntity :


```

class GameplayEntity : public Entity
{
public:
    GameplayEntity(GameManager* mgr) :
        Entity(mgr)
    {
        m_graphics = CreateComponent<GraphicsComponent>();
        m_hitbox = CreateComponent<PhysicsComponent>();
        m_move = CreateComponent<MoveComponent>();
    }

    ~GameplayEntity()
    {
        DestroyComponent(m_graphics);
        DestroyComponent(m_hitbox);
        DestroyComponent(m_move);
    }

private:
    GraphicsComponent* m_graphics = nullptr;
    PhysicsComponent* m_hitbox = nullptr;
    MoveComponent* m_move = nullptr;
};

```

Cela fonctionne, mais ça commence à faire pas mal de code.

De plus, il y a des types de composants que nous n'avons pas encore codé.

Et là vous devez vous demander... Est-ce qu'on doit coder un service pour tous les types de component ?

Et bien... pour ne pas violer le principe de responsabilité unique, ce serait bien !

Mais ça nous oblige à taper beaucoup de code pour pas grand-chose !

On peut peut-être utiliser les templates et les macros préprocesseur pour nous aider dans cette tâche.

**Attention, templates et macros dans la même phrase, à partir d'ici ça commence à devenir velu !
(quoi, ça ne l'était pas déjà ??)**



Etape 12 : Refactos, templates et macros

Pour commencer :

On va partir du principe qu'un **Service** va toujours nous servir de « **fournisseur de composants** ». Ainsi, on encapsule l'utilisation de l'object pool de composants dans un service, et on peut même étendre ce service si nécessaire.

(Imaginez : le service Graphics va bien sûr faire autre chose que juste stocker des composants. Mais un hypothétique « Service HP » ? Pas sûr qu'il serve à grand-chose d'autre...)

Le **GraphicsService** est un service qui a un object pool de **GraphicsComponents**.

Le **HPService** sera un service qui a un object pool de **HealthComponents**.

Il y a là une analogie évidente... Il faut essayer de **factoriser** ça correctement.

Je propose : créez une classe **ComponentService** qui servira de base pour n'importe quel service qui propose des composants. Ce sera une classe **template** qui prendra en paramètres template :

- Le type de component qu'elle fournit
- La taille du pool en size_t

Il s'agit grosso modo de copier-coller votre code actuel du GraphicsService pour en faire un template fonctionnant avec n'importe quel type de Component, et pas juste les graphics components.

Voici un hint sur ce que vous devez faire :

```
template <class CpntType, size_t PoolSize>
class ComponentService
{
public:
    using ComponentType = CpntType;
};
```

```
class GraphicsService : public ComponentService<GraphicsComponent, 128>
{
};
```

Du coup pour le moment, oui votre GraphicsService devient complètement vide. Et oui, ce « using ComponentType = CpntType ; » est important.

La définition de la class GraphicsService, c'est déjà trop à taper quand on sait qu'on va encore avoir beaucoup de services similaires.

Nous pouvons nous servir d'une macro C pour « générer » du code pour nous : on aimerait avoir une macro **DECLARE_COMPONENT_SERVICE** qui taperait tout le « class Graphics Service : public Component blabla... » pour nous.

Elle devrait ressembler à ça :

```
#define DECLARE_COMPONENT_SERVICE(ServiceType, ComponentType, ComponentPoolSize) \
```

Et pouvoir être utilisée comme ça :

```
DECLARE_COMPONENT_SERVICE(HPService, HealthComponent, 64);
```

Hint : pour faire une macro multi-ligne, n'oubliez pas de mettre des « \ » à la fin de chaque ligne pour dire au préprocesseur que la macro continue à la ligne du dessous !

N'utilisez pas cette macro pour le GraphicsService car il servira à autre chose ; par contre, elle semble adaptée pour déclarer le HPService par exemple...

Ensuite, une fois que notre macro fonctionne, attaquons-nous au Game Manager.

On aimerait factoriser à l'aide d'une macro :

- Les **CreateComponent** et **DestroyComponent** explicites avec le type de component que propose notre service (par exemple : GraphicsComponent* CreateComponent() et **DestroyComponent**(GraphicsComponent*) dans le cas de notre GraphicsService).
- La déclaration de la variable **Service** (c'est-à-dire : la ligne **GraphicsService m_graphicsService**).

Tout ceci est rendu possible par les macros. J'aimerais qu'on code une macro **IMPLEMENT_SERVICE** qui fait ça.

Je vous donne un exemple : pour la classe HPService, on aimerait que la macro génère automatiquement pour nous le code suivant :

```

IMPLEMENT_SERVICE(HPService)

=

public:
HealthComponent* CreateComponent()
{
    return m_HPService.CreateComponent() ;
}

private :
template <>
void DestroyComponent(HealthComponent* cpnt)
{
    return m_HPService.RecycleComponent(cpnt);
}

HPService    m_HPService;

```

Servez-vous d'une macro multi-ligne pour faire ça.

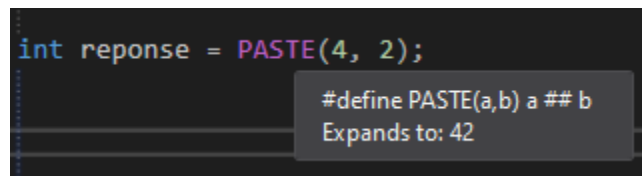
Hint : vous allez avoir besoin de ce qu'on appelle **le token-pasting operator** des macros : `##`

Grosso modo, quand vous voyez « `##` » dans une macro, cela signifie qu'on souhaite « coller » deux représentations textuelles ensemble.

Si par exemple j'ai :

```
#define PASTE(a, b) a ## b
```

Je pourrai faire :



```

int reponse = PASTE(4, 2);

```

#define PASTE(a,b) a ## b
Expands to: 42

Vous allez devoir vous en servir pour écrire des choses dans votre macro.

Autre hint : comprenez-vous désormais l'utilité du « `using ComponentType = CpntType` » public déclaré plus haut dans le service ? Comprenez-vous comment nous pourrions nous en servir ici ?

Au final, votre GameManager ressemblera à quelque chose comme ça :

```
class GameManager
{
public:
    template <typename T>
    T* CreateComponent()
    {
        return CreateComponent<T>();
    }

    template <typename T>
    void DestroyComponent(T* cpnt)
    {
        if (cpnt == nullptr)
        {
            return;
        }

        DestroyComponent_Impl<T>(cpnt);
    }

private:
    template <typename T>
    void DestroyComponent_Impl(T* cpnt) = delete;

    IMPLEMENT_SERVICE(GraphicsService);
    IMPLEMENT_SERVICE(HPService);
};
```

C'est un peu de la magie, mais cela nous simplifiera grandement la vie lorsque nous voudrons implémenter des nouveaux services, nous aurons seulement 2 lignes à écrire :

- DECLARE_COMPONENT_SERVICE(MonNouveauService, MonNouveauComponent, PoolSize)
- IMPLEMENT_SERVICE dans le GameManager.

Etape 13 : Components weak ptrs

You're weak Sasuke.. you lack seasoning



Quand y en a plus, y en a encore !

On aimerait ajouter une dernière chose à notre jeune système d'entité/component :

Je n'aime pas ce code :

```
class GameplayEntity : public Entity
{
public:
    GameplayEntity(GameManager* mgr) :
        Entity(mgr)
    {
        m_graphics = CreateComponent<GraphicsComponent>();
        m_hitbox = CreateComponent<PhysicsComponent>();
        m_move = CreateComponent<MoveComponent>();
    }

    ~GameplayEntity()
    {
        DestroyComponent(m_graphics);
        DestroyComponent(m_hitbox);
        DestroyComponent(m_move);
    }

private:
    GraphicsComponent* m_graphics = nullptr;
    PhysicsComponent* m_hitbox = nullptr;
    MoveComponent* m_move = nullptr;
};
```

Car il casse le principe RAII du C++. Normalement, nous devrions faire les initialisations dans le constructeur et les destructions dans le destructeur.

Le problème avec cette approche est que lorsqu'on souhaite ajouter un nouveau component...

- Il est très facile d'oublier le CreateComponent
- Il est très facile d'oublier le DestroyComponent
- Voire les deux !

Malheureusement, comme ce sont des pointeurs, quand on passe dans le destructeur de l'entité, le dtor n'ira pas destroy le component pour nous.

Alors on aimerait avoir une structure qui gère ça pour nous pour ne pas qu'on oublie.

Nous pouvons nous coder une version simplifiée de ce qu'on appelle un « weak pointer » : C'est en gros une forme de « smart pointer » qui va venir encapsuler un pointeur normal et penser à appeler les delete pour nous. Mais c'est un weak pointer (« pointeur faible »), car il ne garantit pas du tout que la ressource pointée ne sera pas détruite avant qu'on arrête de se servir du pointeur. Pour plus d'infos, cherchez sur les trois smart pointeurs de la std : `unique_ptr`, `shared_ptr`, `weak_ptr`.

Nous nous voulons simplement un type **ComponentWeakPtr** templaté sur un type de component qui prendra en paramètre un `GameManager*` et qui allouera et désallouera pour nous un nouveau pointeur sur component.

Un problème va se poser dans le destructeur : il va falloir utiliser le pointeur sur le manager pour appeler `DestroyComponent`. Donc stockez dans le weak pointer le pointeur sur manager. C'est un peu nullos, parce qu'on a déjà un pointeur sur manager dans l'Entity, mais c'est un surcoût relativement faible de 8 octets qu'on est prêt à payer. Alternativement, vous pourriez essayer de stocker un pointeur sur le `GameManager` dans la classe `Component`. Ainsi, vous éviterez ce surcoût.

Le **ComponentWeakPtr** devra remplir le contrat suivant :

- Templaté sur un type **TComponent**
- Garde en membre un **TComponent*** initialisé à `nullptr`, puis alloué depuis le game manager
- Définit une Rule of Three : destructeur, move constructor, move assignment operator, c'est-à-dire **ComponentWeakPtr(ComponentWeakPtr&& other)** et **ComponentWeakPtr& operator=(ComponentWeakPtr&& rhs)**
- Avec bien sûr un constructeur qui prend en paramètre un **GameManager** pour initialiser le pointeur sur **TComponent**
- Et un constructeur par défaut **delete** (on ne veut pas pouvoir oublier de l'initialiser)
- **Pas de constructeur par copie ni d'opérateur = normal (on ne veut pas de shallow copy)**
- Le move assignment operator devra utiliser le **copy-and-swap idiom** : vous allez devoir implémenter une méthode **void swap(ComponentWeakPtr & other)**.
- Implémente des **surcharges** const et non const de **l'opérateur->** pour accéder directement au component
- Implément une surcharge de **opérateur TComponent*()** const et non const pour pouvoir être considéré comme un `TComponent*`

Ce genre de code devrait fonctionner sans problème :

```
GameManager mgr;

{
    // ComponentWeakPtr<GraphicsComponent> empty;

    ComponentWeakPtr<GraphicsComponent> graph(&mgr);

    ComponentWeakPtr<GraphicsComponent> thief(std::move(graph));

    ComponentWeakPtr<GraphicsComponent> anotherThief(&mgr);

    anotherThief = std::move(thief);

    // ComponentWeakPtr<GraphicsComponent> grou; // must not compile
    //grou = thief; // must not compile
    //ComponentWeakPtr<GraphicsComponent> grou(thief); // must not compile
}
```

Je vous mets un print pour comprendre ce qui se passe :

```
Allocate new cpnt 000002240A7B4320 in ctor
Move copy ctor from 0000000000000000 and 000002240A7B4320
Allocate new cpnt 000002240A7B4340 in ctor
Move operator= from 000002240A7B4340 and 000002240A7B4320
Move copy ctor from 0000000000000000 and 000002240A7B4320
Swap between 000002240A7B4340 and 000002240A7B4320
DeAllocate cpnt 000002240A7B4340 in dtor
DeAllocate cpnt 000002240A7B4320 in dtor
```

Et donc, vous vous en servirez comme ça (c'est un exemple bidon) :

```
class GameplayEntity : public Entity
{
public:
    GameplayEntity(GameManager* mgr) :
        Entity(mgr), m_graphics(mgr)
    {
    }

    ~GameplayEntity()
    {
    }

private:
    ComponentWeakPtr<GraphicsComponent> m_graphics;
};
```


Le souci du **ComponentWeakPtr** c'est que désormais, il nous **interdit** d'utiliser le constructeur de copie ou l'opérateur = régulier de toute entité qui l'utilise, puisqu'on ne peut pas le copier (ça ferait une shallow copy).

Si à un moment vous avez besoin de copier une entité pour initialiser une autre entité, ça ne compilera plus.

Je ne pense pas que ce soit obligatoire, mais si vous voulez régler ce souci, vous allez devoir implémenter un mécanisme de **Clone** pour vos composants.

Vous devrez peut-être réfléchir un peu, et étoffer votre implémentation du Component (par exemple stocker un pointeur dans le component vers le Service d'où il vient, afin d'être capable, sur un Clone(), de demander au service de sortir un nouveau component de l'object pool puis de copier les données).

Je vous laisse méditer là-dessus.