# INTRODUCTION

Dans le monde du jeu vidéo, la vitesse d'exécution est importante afin d'avoir un jeu fluide et rapide. Lorsqu'on créer un jeu, on aime avoir de la diversité et du dynamisme au niveau des 'GameObjects' (dans notre API nous parlerons d’Entités) pour que le jeu soit attrayant. C'est donc dans ce sens que notre API sera développée. Il devra être conçu sur une architecture permettant de travailler rapidement sur les éléments du jeu (éléments regroupés en mémoire contiguë) et une liberté au niveau de la création et l'utilisation des Entités (abstraction des gameobjects).

Pour offrir une flexibilité à l'utilisateur de notre API, il nous faut explorer un paradigme différent de l'Orienté Objet car celui-ci, trop rigide, n’est malheureusement pas adapté pour plusieurs raisons, notamment :

### 1 - Architecture par héritage et dynamisme

Dans un jeu, une entité représente tout objet du jeu imaginable : le personnage, un PNJ, l'arbre où pousse les pommes récoltables, le coffre que tu ouvres, le monstre que tu tues, l'objet sur le sol que tu ramasse, l'animation dans le coin de l’écran. Ces entités ont tous des différences, mais aussi des similarités. En concevant, on peut se rendre compte que nous arrivons à un point ou l'architecture des classes devient archaïque et on se perd dans la multitude d'héritage.

Prenons par exemple un personnage :

- il peut être humain ou elfe  
- bon ou mauvais  
- avoir une faction (voleur ou chevalier pour l'humain ainsi que archer et guérisseur pour l'elfe)

On pourrait imaginer une classe finale qui ressemblerait à :

class Chevalier : Personnage() , Humain(), Ange()  
{ ... }

Et ce n'est pas fini il peut se déplacer, est affichable ...

class Chevalier : Personnage() , Humain() , Ange(), Deplacable(), Affichable(), Cliquable() ..  
{ ... }

Quelle classe fait quoi ? Pour comprendre le fonctionnement final du chevalier il faut trouver les  
définitions de chacune des classes. Il y a des risques de doublons d'informations et pour un nouveau venu sur le projet c'est dur à lire.

Et pour le dynamisme ... si à un moment je veux donner la possibilité à mon chevalier d'avoir le don de guérison de l'elfe ? Je vous laisse imaginer la classe finale et les modifications à apporter à l'architecture en héritage des classes ....

### 2 - Héritage en diamant

Imaginons maintenant une arme à donner à notre personnage. Dans notre jeu, nous aurons des armes à une main (hache) et des armes à deux mains (lance). Et si je veux une hallebarde ? (Une lance avec le bout en hache) Le premier réflexe serait de créer une nouvelle classe Hallebarde qui héritera de hache et lance. Au moment de la compilation, il y aura ambiguïté : est-ce que la nouvelle arme sera utilisé à une main ou deux mains ? Un programmeur non avertis aura du mal à trouver cette erreur à travers cette hiérarchie de classe imbuvable.

# NOTRE API

## CORE

C'est pourquoi notre API se tourne vers un paradigme différent de l'Orienté Objet : L'Orienté donnée à travers ce qu'on appelle 'entité-composant-système' (ou ecs).

Dans un ecs, l'entité est découpée en trois parties :

**Entité** : L'entité elle-même ( une classe avec un simple id numérique qui identifie notre entité )  
  
**Composant** : Ce qui définit l'entité , ses attributs et comportements qui le compose

par exemple : -composant graphique et position pour l'affichage ,  
-composant vélocité et masse pour la gravité et déplacements  
-composant cliquable pour les objet cliquable

-composant loup pour un entité d'une meute de loup.

**Système** : Les 'fonctions' qu'on aurait mises dans les Game Object se retrouvent ici. Toutes les opérations sur les composants sont fait dans les systèmes par exemple :

**- le système graphique** récupère la position et le graphique de tous les entités afin de les afficher.

- **le système physique** prend les composants vélocité et position afin d'y appliquer le déplacement

**- le système d'évènements** boucle sur les évènements qui se sont produit afin de lancer l'action correspondant

- **le système** 'meute de loup' récupèrera la vélocité des entités ayant le composant loup afin qu'ils se déplacent en groupe.

Les Entités de notre API sont donc maintenant construites et définies en trois étape :

- Création de l'entité unique

- Création et ajout des composant a l'entité

- Création du système travaillant sur un type de composant

Afin d'offrir la puissance de cette méthode de création, il est important de laisser la liberté à l'utilisateur de notre API de créer ses propres composants ainsi que ses propres systèmes qui travaillent sur ces composants. Le cœur de notre API devra donc être le plus abstrait possible en ne s'occupant pas de ce que sont les composants et ce que font les systèmes.

En Plus d'offrir cette gestion abstraite, notre API veut décharger l'utilisateur de certaines tâches qui se retrouvent dans la majorité (sinon tous) des jeux vidéo en offrant une couche plus concrète : graphique, physique et évènements. Dans tous les jeux on retrouve des images, qui bougent et qui sont cliquable ainsi que des textes. Lors du déplacement, il faut savoir si l'entité qui se déplace peut se déplacer et qu'il ne rentre pas dans un mur par exemple. Nous offrons donc aussi la gestion des collisions tout en laissant la liberté à l'utilisateur de définir l'Action à exécuter lors de la collision. Concernant les évènements et actions, ils seront détaillés dans la partie système évent.

### Requis fonctionnels du Coré de l'API :

- Création d'une Entité et définition d'une entité à travers des composants - Création d'un type de composant - Création d'un nouveau système - Stockage des entités - Stockage des composants selon leur type - Stockage des système - Mise à jour des systèmes

### Requis non-fonctionnels du Coré de l'API :

- Doit s'abstraire de la composition d'un composant - Doit s'abstraire des opérations d'un système - Implémentation des composants de base : Graphique, Position, Vélocité, Texte, Size

## SYSTEME GRAPHIQUE

Le Système Graphique s'occupe de l'affichage. Étant la partie gourmande d'un jeu, il est important que cette partie soit optimale dans ses opérations. Il est important d'opérer que sur les composants graphiques qui sont à afficher. Par exemple, même si les composants graphiques des boutons du menu existent pendant l'exécution du jeu, nous ne voulons pas que ces composants soient affichés pendant le jeu. C'est pourquoi nous séparons le jeu en Scène. Une scène est simplement une liste des entités qui la compose. Ainsi, à chaque boucle du jeu le système graphique peut boucler sur cette liste et récupérer leur composants graphiques et positions afin d'afficher que les entités nécessaires.

En plus de s'assurer d'opérer que sur les composants nécessaires, même s'ils existent à tout moment, nous ne voulons pas que les images soient chargés en mémoires à tout moment. C'est pourquoi un composant graphique devrait contenir sa source afin que le système puisse les chargé et déchargé. Pour ce faire, le système graphique offre une Action concrète qui s'occupe du changement de scène. Elle doit décharger les images de la scène courante et chargé ceux de la nouvelle scène. Cette Action pourra être lié à n'importe quel évènement. (Clic sur un bouton, mort du personnage.)

Afin de gérer les images et textures du jeu et ouvrir une fenêtre dans laquelle les dessiner, notre API utilisera une librairie externe : sfml.net. Cette librairie est une librairie multimédia aidant la créations de jeux vidéo. Il offre un framework graphique , audio et network. Pour cette version de projet de synthese , nous n'utiliserons que le framework graphique.

Dans ce framework , on y trouve la classe RenderWindow qui s'occupe d'ouvrir une fenetre graphique. Nous avons la possibilité d'effacer l'ecran ( clear ) , ajouter une image a dessiner ( draw ) et afficher les image ajouté ( display ). C'est avec cette suite de fonction qu'à chaque tour de jeu nous créons une illusion de déplacement et animations.

Elle offre aussi la fonction DispatchEvents (). Cette fonction lance les évènements de la fenêtre à partir d'un Délègue vers un évent Handler : Clic de souris, déplacement de souris, déplacement de la fenêtre, changement de grosseur, fermeture de la fenêtre et puis appuie sur une touche. Ces évènements devront être récupérer par le système d'évènements afin d'effectuer les opérations nécessaires suite à la capture d'un de ces évènements. C'est pourquoi le système graphique sera ajouter le premier au Core du jeu afin qu'il soit traité le premier.

Comme la scène sert à séparer ce qui doit être traité ou pas, c'est aussi dans cette classe que nous définirons quelle action est associé à quelle touche du clavier. Par exemple, la touche 'escape' ne doit pas avoir la même signification dans la scène 'menu' et dans la scène de jeu. Nous aurons donc besoin d'un mappage touche -> évènement dans la classe scène. L'évènement est un évènement de clic défini dans la section système d'évènement. Les touches nous viennent de la dépendance sfml.net. Nous avons accès a la classe Keyboard. Key dans laquelle sont définis tous les touches que lance le DispatchEvents () lors de l'appui d'une touche.

### Requis fonctionnels du système graphique

- Ajouter une scène en référence  
- Récupérer la scène actuelle ( afin que les autre systèmes ne s'occupe que des entités pertinentes aussi )

### Requis non-fonctionnels du système graphique

- Implémentation de la classe Scène  
- Implémentation de l'Action concrète de changement de scène  
- Affichage du maximum d'entité dans une scène en restant au-dessus d'un minimum de 40 images par secondes  
- Dépend de sfml.net et Csl

## SYSTEME D'EVENEMENTS

Le système d'évènements est celui qui s'occupe de traiter les évènements qui se sont produit pendant un tour de boucle du jeu. Il doit s'abstraire de ce qu'est l'évènement qui s'est produit ainsi que ce que fait l'action à lancé. Son fonctionnement est : Il s'est passé un évènement, exécutons son action. Dans les évènements d'un jeu on peut compter les évènements de l'utilisateur (clic de souris, appuie et relâchement d'une touche) et les évènements lancé d'un système, par exemple une collision envoyée par le système physique ou le système de gestion de ressource qui avertis que le blé est prêt à être récolté.

C'est à travers ce système d'évènement et d'action que nous permettons à l'utilisateur de l'API d'ajouter de l'interaction à ses entités et systèmes. Ce système offre donc la classe 'GameEvent' et l'interface 'Action'. Chaque GameEvent doit être lié à une Action afin que lorsque le système d'évènements reçoit un évènement, il peut lancer l'action qui lui appartient.

Le système d'évènements fonctionne en 2 parties.

1 - Afin de ne pas lancer des actions pendant les opérations d'un autre système (question de rapidité d’exécution) nous devons d'abord récupérer les évènements (clic de souris, appuie d'une touche et évènements lancé d'un autre système) puis nous les ajoutons à une liste afin d'être traité pendant la mise à jour du système d'évènements

2 - Lors de la mise à jour du système d’évènements, nous bouclons sur les évènements qui se sont produits et nous lançons l’action qui lui est lié.

La détection d'un clic, l'appui d'une touche et son relâchement sont des évènements typiques dans un jeu que nous offrons dans cette partie de l'API. Pour le clic, ceci se fait à travers un composant 'cliquable' et un évènement de clic. L'action à réaliser dans l'évènement de clic est au choix du créateur.

Pour ce qui est de l'appui d'une touche, il était important de séparer quel évènement est lancé sur l'appui d'une touche c’est pour cette raison que le mappage touche-évènement est fait dans la scène. Ainsi, lors de l'appui d'une touche, le système d'évènement interroge le système graphique et la scène courante si un évènement est lié à la touche avant d'ajouter l’évènement à la liste.

### Requis fonctionnels du système d'évènements

- Création d'évènement - Création d'action à relié aux évènement - Réception d'un évènement à partir d'un autre système (Notifier / Subsidier pattern)

### Requis non-fonctionnels du système d'évènements

- Faire le minimum d’Operations lors de la réception d'un évènement réel (clic de souris, touche, autre système) - Exécuter les opérations lors de la mise à jour du système d'évènements - Se renseigner sur la vitesse d'update optimale - S'abstraire de quel évènement a été lancé et ce que fait l'action. (Command Pattern) - S'occuper de récupérer les évènements reçus de la fenêtre sms (RenderWindow) envoyer par le DispatchEvents () du moteur graphique.