Sangchanmahola Erwan

Rapport projet Space Invaders

Présentation du sujet :

J'ai décidé de partir sur la piste noire avec le modèle entité composant système après avoir fini rapidement la piste verte. Le premier problème a été de définir toute l'architecture ecs du code pour avoir quelque chose de fonctionnel. Puis d'implémenter les fonctions nécessaires à un Space Invader.

Les fonctionnalités disponibles sont celles dans la piste verte plus une musique de fond (dont je n'ai pas réussi à régler le volume).

La musique est désactivable en mettant la variable `playSound` a false dans SoundSystem.cs.

Structure du programme :

La structure du programme est une structure ecs, les entités contiennent des composants qui détermine leur fonctionnement et un système traitent les entités qui contiennent les composants requis.

La classe principale est le gameEngine qui permet de faire le lien entre EntityManager et SystemManager.

EntityManager est la classe qui s'occupe d'initialiser et de garder les Entity du jeu dans une HashSet

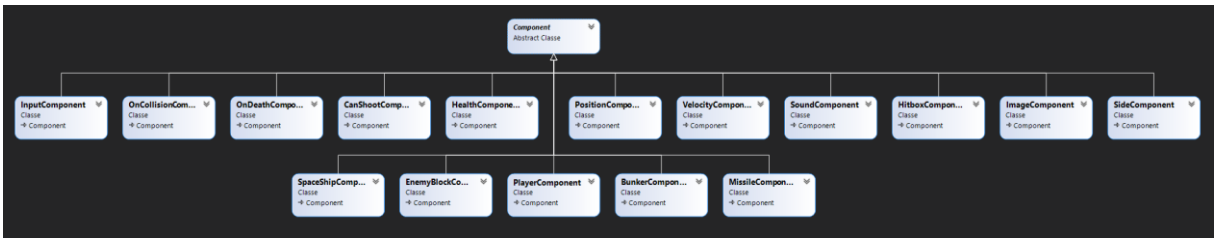
Une Entity est un objet qui contient des Component.

Les Component permettent d'avoir des informations sur l'Entity ex : la position.

SystemManager est la classe qui regroupe tous les System dans une List.

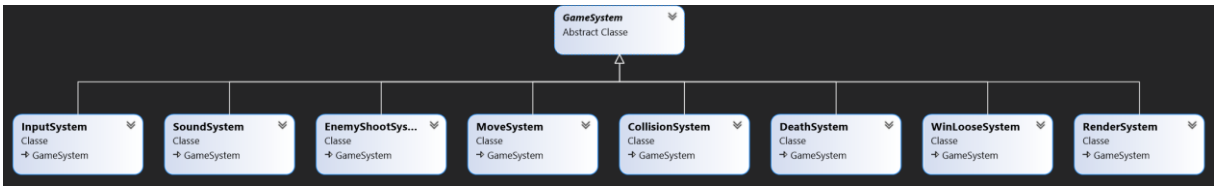
Un System permet de s'occuper de la partie logique du jeu ex : collision.

Liste des composants :



Liste des systemes :

Sangchanmahola Erwan



La creation du jeu s'effectue en plusieurs étapes.

1. On initialise toutes les entités du jeu en leurs associant des composants dans l'EntityManager.
2. On initialise tous les systemes dans le SystemManager.

Une fois le jeu lancé, la classe gameEngine va appeler la méthode update du SystemManager qui va actualiser tous ses systèmes dans l'ordre ; input, sound, enemyShoot, move, collision, death, winLoose.

Le système render est actualisé indépendamment des autres systemes à chaque paintEvent par la classe gameForm.

Input : permet de gérer les touches du clavier.

Sound : permet de jouer la musique de fond.

EnemyShoot : permet de faire tirer les ennemis.

Move : permet de faire bouger les entités qui peuvent être bougé.

Collision : permet de détecter et de gérer les collisions du jeu.

Death : permet de gérer les entités mortes.

WinLoose : permet d'actualiser l'état du jeu.

Problèmes rencontrés et solutions :

<u>Architecture :</u>

Le premier problème était de définir une architecture de code qui marche et cela m'a pris pas mal de temps pour la théoriser. Au début, je voulais partir sur l'idée d'avoir un groupe de node qui contient des composants pour chaque système mais j'ai abandonnée cette idée car je n'ai pas trouvé de moyen de remonter à l'entité. Par exemple comment retrouver l'entité à partir d'un composant pour le détruire. J'ai donc décidé d'enlever les nodes et de passer directement les entités aux systèmes.

Après, j'ai dû définir les composants Position, Velocity, Image, OnCollision, Health, Side, ... puis je me suis rendu compte que j'ai dû ajouter des composants vide pour faire office de tag pour différencier les types d'entités comme un vaisseau, un bunker ou un missile.

<u>Collision :</u>

La collision a été plus compliqué à mettre en place par le fait qu'il fallait savoir sur quel type d'entité on entrait en collision ainsi que la fonction à appliquer.

Pour cela j'ai créé un type délégué pour stocker et déterminer la fonction à utiliser selon le type de l'entité :

```
//Find collision action
Action<Entity, Entity, int, int> collision = null;
if (collided.GetComponent(typeof(BunkerComponent)) != null) collision = OnCollisionBunker;
if (collided.GetComponent(typeof(MissileComponent)) != null) collision = OnCollisionMissile;
if (collided.GetComponent(typeof(SpaceShipComponent)) != null) collision = OnCollisionSpaceShip;
if (collision == null) return;
```

Et ensuite il suffisait simplement d'appeler la fonction déléguée lorsqu'il y a une collision :

```
if (colliderImageComponent.Image.GetPixel(i, j) != Color.FromArgb(0, 255, 255, 255)
    && collidedImageComponent.Image.GetPixel(x, y) != Color.FromArgb(0, 255, 255, 255))
{
    collision(collider, collided, x, y);
}
```

Tir :

Pour ne pas pouvoir tirer pendant que le missile qui appartient au vaisseau est en vie j'ai décidé de rajouter un composant « canShootComponent » qui permet de savoir si un vaisseau peut tirer ou non. A la création, chaque vaisseau a ce composant et peut donc tirer. Lorsqu'un vaisseau tire ce composant est enlevé et il ne peut donc plus tirer.

Problème : comment remettre le composant une fois détruit. Pour cela, lors de la création d'un missile un composant « OnDeathComponent » est ajouté et il va se charger de remettre le composant a l'entité à sa mort :

```
OnDeathComponent onDeathComponent = new OnDeathComponent(() => entity.addComponent(new CanShootComponent()));
```

Enemy Block :

Le problème est de pouvoir synchroniser le déplacement de tous les vaisseaux avec les changements de direction lorsqu'un vaisseau rencontre un mur.

Pour cela j'ai créé une entité qui a une position, une vitesse, une taille (hitbox) et avec un composant « EnemyBlockComponent » qui permet de le retrouver.

Cette entité permet d'avoir la position et la taille du block d'ennemis et il suffit donc de faire déplacer les ennemis en même temps que le block.

```
if (outOfBonds(gameEngine, enemyBlock))
{
    moveBackEntity(enemyBlock, deltaT);
    enemyBlockVelocity.VelocityX *= -1.02;
    enemyBlockVelocity.VelocityY = 1500;
}
else
{
    moveBackEntity(enemyBlock, deltaT);
    enemyBlockVelocity.VelocityY = 0;
}
foreach (Entity entity in enemyList)
{
    VelocityComponent entityVelocity = (VelocityComponent)entity.GetComponent(typeof(VelocityComponent));
    entityVelocity.VelocityX = enemyBlockVelocity.VelocityX;
    entityVelocity.VelocityY = enemyBlockVelocity.VelocityY;
    entityVelocity.AngularVelocity = enemyBlockVelocity.AngularVelocity;
}
```

Son :

Le problème était de pouvoir implémenter du son dans le jeu. J'ai utilisé la classe SoundPlayer mais je n'ai pas pu mettre une musique d'ambiance avec les son du jeu :

```
HashSet<Entity> soundEntities = gameEngine.entityManager.GetEntities(typeof(SoundComponent));
foreach(Entity entity in soundEntities)
{
    SoundComponent soundComponent = (SoundComponent)entity.GetComponent(typeof(SoundComponent));
    if(soundComponent != null)
    {
        SoundPlayer sound = new SoundPlayer(soundComponent.SoundPath);
        sound.LoadAsync();
        sound.Play();
        entity.removeComponent(typeof(SoundComponent));
    }
}
```

Lorsque le son d'une entité était joué, la musique d'ambiance s'arrêtait. Après quelques recherches, j'ai trouvé que c'était une limitation de la classe SoundPlayer mais je n'ai pas réussi à trouver une alternative. J'ai donc décider de ne garder que la musique de fond.

Sinon l'idée était d'ajouter un composant son lorsqu'un tir de missile est effectué.