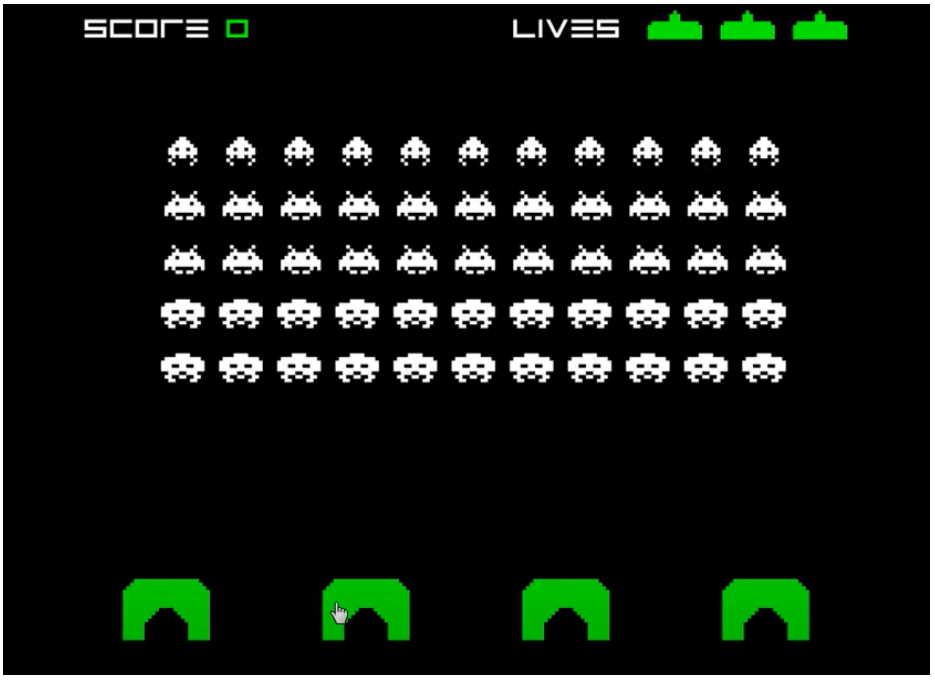
Rapport "Space Invaders"

Rachid ABDOULALIME, Mohamed DJEGHBALA



Sommaire :

1. Description succincte du problème

L'objectif du projet était de développer en C# un jeu rétro se nommant Space Invaders développé par la société japonaise Taito, sorti en 1978 sur borne d'arcade.

Le principe du jeu est de contrôler un vaisseau situé en bas de l'écran se déplaçant horizontalement, tandis qu'une horde de vaisseaux ennemis descend lentement de haut en bas. L'objectif en tant que joueur est d'éliminer tous ces vaisseaux à l'aide d'un missile avant qu'ils n'atteignent la hauteur du vaisseau joueur.

Le joueur dispose de bunkers destructibles qui lui servent d'abri face aux tirs des missiles ennemis et doit éviter les attaques afin de survivre et de remporter la partie. Le défi est que le bloc de vaisseaux ennemis se déplace de plus en plus rapidement et tire plus fréquemment au fur et à mesure que le jeu progresse.

2. Description de la structure du programme

Le programme est construit en suivant les principes fondamentaux de la programmation orientée objet (POO). Cette approche organise le code en utilisant des entités appelées "objets", qui regroupent des données et des méthodes, favorisant ainsi une modélisation réaliste et une réutilisabilité efficace du code.

La structure du programme se compose de plusieurs classes interdépendantes, chacune jouant un rôle spécifique dans la mise en œuvre du jeu Space Invaders :

GameObject : Cette classe est une entité abstraite représentant un objet actif du jeu. Elle définit des méthodes que tous les objets du jeu partageront. Cette classe abstraite fournit une base commune pour d'autres classes spécialisées.

SimpleObject : Une classe abstraite qui hérite de GameObject. Elle encapsule les caractéristiques communes à plusieurs objets du jeu, telles que la position, la taille, l'image représentative, et le nombre de vies.

Vecteur2D : Une classe qui permet la gestion des coordonnées et des déplacements dans le jeu. Elle est utilisée pour représenter les positions des objets et faciliter les calculs liés aux déplacements.

SpaceShip : Une classe héritant de SimpleObject, représentant les différents vaisseaux du jeu. Elle possède des caractéristiques spécifiques, telles que la vitesse de déplacement et le nombre de vies.

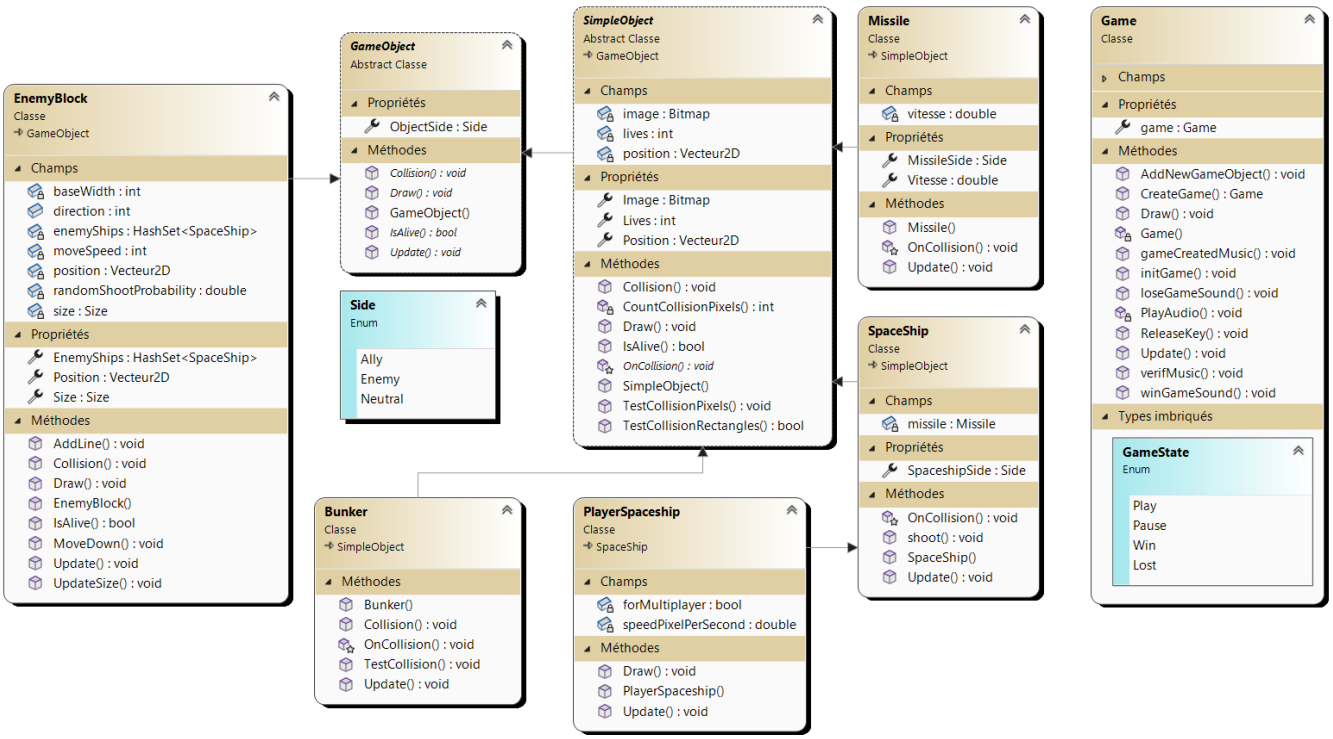
Missile : Une classe héritant de SimpleObject, permettant aux vaisseaux joueurs et ennemis de tirer des missiles verticalement. Elle gère les déplacements des missiles et leur interaction avec d'autres objets.

PlayerSpaceShip : Une sous-classe de SpaceShip qui représente les vaisseaux joueurs. Elle inclut des fonctionnalités spécifiques au joueur, comme la gestion des commandes de déplacement.

Bunker : Une classe héritant de SimpleObject, représentant les bunkers du jeu. Les bunkers sont des éléments statiques qui offrent un abri au joueur contre les tirs ennemis.

EnemyBlock : Une classe héritant de GameObject, représentant le bloc de vaisseaux ennemis. Cette classe gère un groupe de vaisseaux ennemis, leur déplacement, leurs tirs, et leur interaction avec d'autres objets.

Voici l'UML de notre projet :



3. Présentation des problèmes soulevé

➔ **Reset une partie :**

Une des fonctionnalités sur laquelle je bloquais était de pouvoir relancer une partie, et je stagnais dessus sur plusieurs séances. Le blocage venait du fait que je ne savais pas de quelle manière je pouvais relancer le jeu. Un des recours était de voir des idées de sources externes telles que StackOverflow ou Youtube. Or, les idées observées n'étaient pas utiles car souvent les devs avaient déjà des méthodes permettant de reset une partie, faisant reset juste en appelant ces derniers..

Par la suite, j'avais pensé à retirer les composants du jeu de la liste du Game et de la vider puis appeler la méthode createGame(), chose qui ne fonctionnait pas. Ainsi, je suis parvenu à créer une méthode initGame() qui contenait le bout de code du constructeur en l'ayant retiré et qui est appelé dans le constructeur mais aussi lorsque la partie se termine (soit lorsque la touche espace est pressée). Et donc finalement, j'ai adopté cette façon car elle optimise le code puisque la méthode fait maintenant deux choses à la fois, créer une partie lorsque le jeu démarre et relancer une partie finie.

➔ **Audio sonore :**

La classe de C# System.Media présentait une limitation, car elle ne permettait de jouer qu'un seul son à la fois et ne présentait pas . Ainsi, il était impossible de jouer simultanément le son du jeu principal en boucle ainsi que les autres sons (victoire, défaite, laser, et explosion).

Pour résoudre cette contrainte, une solution a été mise en place en installant un package NuGet appelé NAudio, donnant accès à des méthodes telles que Play(), Pause() et Stop() afin de gérer correctement la sonorité du jeu en fonction des différents états. Ce qui nous a facilité la tâche sans avoir à coder nous même ces méthodes citées, tout cela en ayant trouvé un package différent de celui du début.