Rapport Space Invaders

Projet C# - POO - Théo SZANTO

Présentation du projet

Le projet consiste en la création d'un jeu vidéo en 2D à l'aide du langage C#. Le jeu en question est un Space Invaders : le but est de détruire des ennemis qui arrivent par le haut de l'écran sans se faire détruire soi-même. C'est un grand classique des bornes d'arcades, possédant de nombreuses variantes et extensions possibles, mais le sujet ici était très sommaire. Le pattern Entité / Composant / Système a été adopté ici.

Cahier des charges

Le joueur

Le joueur possède un vaisseau avec un nombre de vie affiché sur l'écran. Il peut se déplacer à droite et à gauche et tirer des missiles vers le haut.

Les ennemis

Les ennemis sont organisés en un bloc de plusieurs lignes, et se déplacent de droite à gauche. Le bloc descend progressivement, et accélère au fil du temps. Chaque ennemi a une chance de tirer un missile vers le bas qui augmente au fil de sa progression. Comme le joueur, ils possèdent un nombre de vie, et leur objectif est d'atteindre le joueur avant d'être détruit.

Les missiles

Tous les vaisseaux peuvent tirer des missiles pouvant entrer en collision avec les objets du jeu.

Collision avec les autres missiles

Lorsque deux missiles entrent en collision, ils se détruisent mutuellement, indépendamment de la vie de chacun.

Collision avec les bunkers

Lorsqu'un missile touche un bunker, il supprime les pixels avec lesquels il est entré en collision et il perd autant de vie que de pixels supprimés.

Collision avec les vaisseaux

De manière similaire aux bunkers, il enlève des pixels et perd de la vie, mais peut survivre et continuer sa route s'il avait plus de vie que le vaisseau en question.

Les bunkers

Le joueur peut se cacher derrière des protections, mais il ne pourra pas tirer au travers. Ces éléments sont impactés par les missiles.

Le déroulement de la partie

Le joueur gagne dans le cas où il détruit tous les assaillants, et perd si il est détruit ou qu'un vaisseau ennemi atteint l'objectif. Il est alors possible de relancer une nouvelle partie. Le jeu peut être mis en pause, interrompant toutes les actions.

Base de code

Un code de base à été fourni, mais n'était pas adapté au pattern ECS, et a donc été quasiment intégralement supprimé / remplacé. Les seules classes ayant survécu sont `Program` et `GameForm` (ayant toutefois subi des modifications conséquentes).

Résultat

Le projet a été terminé et est fonctionnel, disponible à travers le dépôt de source GitHub suivant : github.com/indyteo/SpaceInvaders.

Une courte vidéo de démonstration du jeu est disponible à l'URL : theoszanto.fr/-/SpaceInvaders.mp4.

Réalisation du projet

Le code est architecturé selon le design pattern ECS, ce qui implique la présence de nombreuses classes pour représenter les composants et les nœuds, ainsi que les systèmes.

Engine

Le cœur du jeu est la classe `Engine`, qui possède les `Entity`, les `System` et les `Node`. Il est capable de créer les nœuds à partir des entités, et les systèmes peuvent extraire des références vers les listes de nœuds par type, pour s'adapter en fonction des ajouts et suppressions d'entités.

System

Les systèmes constituent la boucle de jeu. Ils sont exécutés dans un ordre précis par l'`Engine`, et peuvent ou non être suspendus. Il est possible d'ajouter et retirer des systèmes, ayant pour effet de les démarrer et de les éteindre. Chaque `System` peut récupérer des listes de `Node` à travers l'`Engine`, qui sont mises à jour dynamiquement au fil du temps.

`NodeIteratingSystem<Node>`

Cette sous-classe de `System` permet de facilement itérer sur une liste d'un sous-type de `Node` donné. La méthode `Update` est alors automatiquement implémentée pour appeler la méthode `UpdateNode` sur chaque élément de la liste.

`SystemOrder`

Cette énumération de valeurs représente l'ordre dans lequel les systèmes doivent s'exécuter. Les valeurs possibles sont : `BEFORE`, `PRE_UPDATE`, `UPDATE`, `POST_UPDATE`, `BETWEEN`, `PRE_RENDER`, `RENDER`, `POST_RENDER` et `AFTER`.

Liste des systèmes

`GameControlSystem`

Ce système gère le lancement d'une partie, ainsi que la fin (victoire ou défaite). Il utilise `GameNode` et s'exécute en position `BEFORE`.

`ChildrenGraveyardSystem`

Ce système observe la liste d'entités de l'`Engine` pour supprimer les entités enfants lorsqu'elles sont supprimées. Il utilise `ParentNode` et s'exécute en position `BEFORE`.

`SpaceshipControlSystem`

Ce système permet au joueur de contrôler son vaisseau. Il utilise `SpaceshipControlNode` et s'exécute en position `PRE_UPDATE`.

`LinearMovementSystem`

Ce système déplace les entités de façon linéaire et les détruit lorsqu'elles sortent de l'écran. Il utilise `LinearMovementNode` et s'exécute en position `PRE_UPDATE`.

`EnemyMovementSystem`

Ce système déplace le bloc d'ennemis latéralement avec décalage vers le bas. Il utilise `EnemyMovementNode` et s'exécute en position `PRE_UPDATE`.

`CollisionSystem`

Ce système gère la collision entre les entités. Il utilise toutes les `CollisionNode` et s'exécute en position `UPDATE`.

`EnemyShootSystem`

Ce système permet aux ennemis de tirer de façon aléatoire. Il utilise `EnemyShootNode` et s'exécute en position `POST_UPDATE`.

`RenderSystem`

Ce système affiche les entités sur la fenêtre. Il utilise `RenderNode` et s'exécute en position `RENDER`.

`GameOverlaySystem`

Ce système affiche l'interface du jeu (nombre de vie, texte "pause"). Il utilise `GameNode` et s'exécute en position `POST_RENDER`.

`GrimReaperSystem`

Ce système supprime les entités dont la vie est inférieure ou égale à 0. Il utilise `SoulNode` et s'exécute en position `AFTER`.

`PauseControlSystem`

Ce système permet de contrôler la pause. Il utilise `GameNode` et s'exécute en position `AFTER`.

Node

Les `Node` regroupent des `Component` et sont construits selon les `Entity`. Ils sont utilisés par les `System` sous forme de listes à travers l'`Engine`, et permettent de donner des comportement aux entités.

Component

Les composants sont des conteneurs de valeurs partagés entre les nœuds d'une entité.

Entity

Les entités sont composés d'une liste de `Component`, et d'un type permettant de savoir de quels `Node` elles ont besoin. Les entités ont également un identifiant sous la forme d'un auto-incrément pour pouvoir être référencées à travers les `Node`.

EntityCreator

Cette classe utilitaire permet de créer des entités en leur donnant les `Component` nécessaires, et de lister leurs dépendances de `Node`.

Liste des entités

Il existe 6 entités dans l'énumération `EntityType` : `GAME`, `SPACESHIP`, `ENEMY`, `MISSILE`, `BUNKER` et `ENEMY_BLOCK`. Chaque type d'entité possède une liste des types de `Node` dont elle aura besoin.

Problèmes rencontrés

Le seul problème à été du côté du C# qui n'autorise pas l'usage d'un type non-paramétré (comme `List<?>` en Java). Cela a causé un problème de type générique au niveau de la `ClassMapList<T>` qui nécessite ainsi impérativement le type réel des listes à travers `<U>` dans chaque méthode pour pouvoir caster. Ceci empêche donc d'appeler ces méthodes avec un type parent, et donc dans l'`Engine` de manipuler les `Node` de façon générique. La solution de contournement adoptée à été de lister statiquement les sous-types de `Node`. Cette solution n'est pas souple, mais sans pouvoir caster les listes à des types parents (`List<? extends X>` en Java), est inévitable.